SOFTWARE ASSESSMENT STUDY
for the
BUREAU OF LAND MANAGEMENT

DELIVERABLE 6/7.5

FINAL

ADP MODERNIZATION PROJECT

JANUARY, 1988

Contract AA852-CT5-15

American Management Systems, Inc.
1777 North Kent Street
Arlington, Virginia 22209

TABLE OF CONTENTS

APPENDICES

1. INTRODUCTION

1.1 OBJECTIVE AND SCOPE

This document contains the Software Assessment, Deliverable 6/7.2, of the Automated Data and Equipment Modernization Project (ADEMP). It presents an assessment of the condition of BLM's current software applications in terms of usability and supportability as well as the feasibility of a Software Improvement Program (SIP), a form of software preventative maintenance program endorsed by the General Services Administration. BLM can use this document to determine how to handle these applications as they are moved to BLM's new technical architecture and whether an SIP is appropriate.

In particular, this assessment will help BLM respond to OMB Bulletin 87-10, which provides guidance to Federal Agencies concerning how to report their plans for reducing software maintenance obligations by 25% over the next three years. The FY 1986 report on Management of the United States Government stated a goal of 25% reduction in software maintenance obligations government-wide over three years. The Office of Management and Budget (OMB) is requiring agencies to submit plans for reducing maintenance obligations and explain why continued maintenance of an application, particularly older applications (i.e. over ten years old), is justified.

A significant portion of BLM's applications are old. One alternative to rewriting these old applications is to revitalize them with an SIP instead, reducing maintenance costs while preserving much of the effort invested in the original system. A SIP is a comprehensive program for improving software in terms of programming and maintenance efficiencies. A SIP is not an application-specific technique. As such, an agency's entire suite of applications must be considered in determining if an SIP is feasible and desirable.

Although state implemented applications are not within the scope of this study, the SIP approach recommended in this document, and other ensuing

actions taken to review/enhance BLM's application software, can also be implemented in the State offices. The scope of this assessment is to address the Bureau-wide application systems. Although the assessment of State implemented and MOSS based application systems are beyond the scope of this study, the recommendations identified in this document should be considered for these other application systems.

The ADEMP project, which the Software Assessment is designed to support, has a Bureau-wide focus, i.e. those applications which serve more than one BLM State Office or operate on equipment shared among State Offices (i.e. the DSC DPS-8/70). Other than the Automated Resource Data (ARD) applications, these Bureau-wide applications are implemented on the Honeywell computers at DSC, the State Offices and BIFC.

We have not included Automated Resource Data (ARD) applications in the assessment or the SIP feasibility study. (Note: ARD supersedes the Geographic Information System - GIS.) These applications primarily use the public-domain software package, MOSS, with limited BLM developed custom software. The techniques used in a SIP are designed to improve custom in-house software applications not package applications. As BLM will require the support of MOSS as part of the new architecture, no conversion/improvement decision is necessary. This requirement also means a SIP for ARD applications would not be appropriate. Therefore, when we use the term Bureau-wide applications in this document, we mean non-ARD Bureau-wide applications.

1.2 BACKGROUND ON BLM AND ITS ADP ENVIRONMENT

It is assumed that the reader is familiar with the Bureau and its mission. For those unfamiliar with the BLM, please refer to the Functional Requirements document produced as part of Task 1 of the ADEMP project. Currently, BLM's bureau-wide applications for BLM operate on a Honeywell DPS-8/70 at the Denver Service Center (DSC) and on a series of Level 6 Honeywell minicomputers at each of the State Offices and the Boise Interagency Fire Center (BIFC). These computers are connected by star networks in each of the State Offices and at DSC. Each network is administered by the appropriate State Office or DSC staff. (For a detailed description of BLM's technical

environment, please refer to Existing Capabilities, June 1986, Task 2 of the ADEMP project.)

1.2.1 History of BLM Current Bureau-Wide Applications

Bureau-wide ADP applications were initially implemented on a Burroughs computer system BLM shared with the Bureau of Mines and later transferred to a BLM-owned Burroughs computer system installed in the DSC. This computer was replaced by a Honeywell 66 in the late 70's. The Burroughs applications were converted (without any redesign) to run on the Honeywell machine. This machine was later upgraded to the current DPS-8/70. Shortly thereafter Honeywell Level 6 minicomputers were installed in each of the State Offices and at BIFC to support State-specific applications. Thus while DSC has some older applications on the DPS-8/70 that were converted from the Burroughs, the applications implemented on the State Office's Level 6 computers are fairly new and have never been converted.

BLM follows a dual strategy for Bureau-wide applications. The bulk of the Bureau-wide applications ~~requiring data sharing~~ requiring data sharing are developed centrally by DSC for the DPS-8/70 machine and accessed by field staff through the communications network. In addition to these centrally developed applications, BLM also uses a "Lead State" concept. Under this concept, a particular State Office (the "lead state" for that application) develops an application on its Level 6 and then provides copies of that application (and subsequent updates/modifications) to other State Offices for implementation on their Level 6's. This concept has been implemented in a few instances (e.g., MRO, ORCA, etc.) and could conceivably be expanded for future development and maintenance efforts.

1.2.2 Planned Applications and DOI ADP Initiatives

BLM initiated the ADEMP project to provide the technical specifications for an RFP for systems hardware, system software, and data communications to not only support the applications currently running on the Honeywell equipment but also provide the capability necessary for new BLM software development initiatives and to consolidate ADP equipment. These new

applications include the Land Information System (LIS) which is a combination of ALMRS, ARD, and Cadastral Coordinates/GCDB application systems. Some existing applications will be superceded by these new ones. For example, the current Mining Claims application will be subsumed by LIS.

Also affecting the future of BLM's software are initiatives by the Department of the Interior to share administrative software among the Bureau within DOI. The DOI has targeted 4 major areas in which it intends to select a Lead Bureau to service the needs of the other Bureaus within DOI. These areas are: Real property management, financial management, procurement support, and payroll/personnel. These initiatives must be considered when assessing the future of BLM's applications. For example, many existing BLM applications (e.g., General Ledger, Automated Fleet Inventory, Perpetual Inventory) may be fully or partially replaced by DOIs Financial Integrated Review for Management (FIRM). In support of this DOI initiative, a contract has been to implement a departmentwide off-the-shelf accounting system. The methodology to be employed for the implementation of this system (i.e. centralized/partially centralized/decentralized) and the implementation schedule have yet to be defined.

1.3 ASSUMPTIONS AND CONSTRAINTS

The following assumptions were made for the Software Assessment:

- All current applications coded in FORTRAN 66 and COBOL 64 will be upgraded to ANSI FORTRAN 77 and ANSI COBOL 74 respectively, prior to any conversion efforts to the new technical environment.

- BLM has a substantial investment in application software which, should be preserved where justified.

- The interviews of BLM support staff and the current status of change requests reflect the stability of the existing applications and are indicative of how close the applications is to meeting user requirements.

- The new architecture will provide comparable functionality to the functionality provided by the current applications (i.e. no application currently in use will simply disappear. It will be either converted and continued on the bureau-wide facilities, moved to local or PC equipment, or its functions subsumed by other applications).

- The DOI FIRM initiative will replace portions of BLM's applications systems within a short time of (two to three years) or concurrent to the conversion of these applications to the new technical architecture.

1.4 DOCUMENT ORGANIZATION

We have organized this document into six chapters (including this introduction) and four appendices:

- **Chapter 2 Methodology**

  This chapter describes the steps taken during this analysis, the rationale behind each step, and factors used in assessing BLM's applications. It includes brief descriptions of the automated cost estimating model, the COnstructive COst MOdel (COCOMO), and the Software Improvement Program (SIP) methodology used in the analysis. A more detailed description of SIP is provided in Appendix A.

- **Chapter 3, General Software Assessment**

  This chapter describes BLM's current applications in terms of the factors described in Chapter 2 and their probable disposition (e.g., applications which will be replaced by other applications currently under development by BLM, applications that will continue to be used by BLM, etc.). We

also briefly identify and discuss the actions BLM may take for each (e.g., archive, convert, etc.).

- **Chapter 4, Detailed Software Assessment**

  This chapter provides detailed assessments of the nineteen large (in terms of lines of code) existing BLM applications expected to continue when the new technical architecture is implemented.

- **Chapter 5, Feasibility of a Software Improvement Program**

  This chapter examines the feasibility of implementing a SIP as an augmentation to BLM's Life Cycle Management program. Alternative courses of action for BLM's existing applications are presented along with a brief discussion of the advantages and disadvantages of each one.

- **Chapter 6, Ensuing Steps**

  This chapter briefly outlines the steps to be taken by BLM should they decide to implement a SIP. Also included is a discussion of four criteria (size, mission/critically, volatility, and high leverage/return) considered key in determining whether or not an existing application should be upgraded/modernized.

- **Appendix A, Software Improvement Program Description and Examples**

  This Appendix contains a copy of the Guidelines for Planning and Implementing a Software Improvement Program (SIP), prepared by GSA's Federal Conversion Support Center.

- **Appendix B, Sample Software Conversion and Assessment Survey**

  This Appendix contains a sample of the survey forms used to collect software conversion and assessment information on DSC and State Office application systems.

- **Appendix C, Software Assessment Survey Data**

  This Appendix contains summary sheets of the data collected on nineteen large and continuing BLM applications (some of which will be partly subsumed by FIRM)

- **Appendix D, COCOMO Development Cost Estimates**

  This Appendix contains a summary of the output generated by the COCOMO model for the applications examined and a more detailed description of the COCOMO model itself.

- **Appendix E, References**

  This Appendix contains references for the published BLM, GSA, and industry documents used during this analysis. This Appendix does not include references to deliverables produced during the course of the ADEMP project.

2. METHODOLOGY

This chapter describes the methodology used to assess the current condition of BLM's Bureau-wide software and to determine the feasibility of a Software Improvement Program (SIP). It is organized into five sections. The first section describes the Software Improvement Program and our approach to determining the feasibility of a SIP for BLM. The second section describes our approach for the general assessment of Bureau-wide applications. The third section describes the detailed assessment approach taken for those continuing applications (as opposed to applications discontinued or superceded under the new architecture) in which BLM has a significant investment and which are likely to serve BLM for some time to come. The fourth section describes the COnstructive COst MOdel (COCOMO) which was used as part of the detailed assessment analysis to estimate redevelopment costs for BLM's continuing applications. Finally, the fifth section describes the sources for the data used in this analysis.

2.1 SOFTWARE IMPROVEMENT PROGRAM

A Software Improvement Program (SIP) is an incremental and evolutionary approach to modernizing existing software endorsed by GSA's Office of Information Technology, Federal Conversion Support Center (FCSC) which publishes a series of handbooks on how to plan for and execute an SIP. The overall intent of an SIP is to reduce application maintenance cost without necessarily rewriting old applications. The FCSC states in its manual, Guidelines for Planning and Implementing a Software Improvement Program:

> "the goals of a SIP are to improve maintenance and achieve as much functional independence and standardization of interfaces within and across applications as possible. It is a definitive maintenance program for software which is designed to preserve the value of past software investment."

Functional independence refers to the isolation of the functions performed by an application. Once functional independence is achieved, common functions can be shared in modules. This simplifies maintenance, reducing maintenance effort and costs. A copy of the GSA guideline report for SIPs is included in Appendix A.

2.1.1 SIP OBJECTIVE

The objectives of a SIP are to:

- Reduce software maintenance costs and improve control;
- Upgrade software engineering techniques to current or state-of-the-art levels;
- Preserve the value of past software investments; and
- Maximize programmer productivity.

We suggest the SIP methodology be used to augment the systems development methodology currently in use by the BLM Life Cycle Management (LCM) approach. The Bureau's LCM for ADP projects provides direction and guidance for the initiation, development, and implementation of automated data systems as well as maintenance. All the SIP activities occur within the maintenance stage of the LCM process. The SIP coordinates the maintenance stage activities for all the applications included in the SIP to achieve a common goal (e.g. achieve machine independence).

Many organizations have had success implementing SIPs. For example, Raytheon Corporation eliminated 40 to 60% of the redundancy in their software by in their application development using an SIP. San Diego County Department of Education cut maintenance time by 70% by implementing an SIP. SIPs have also been successfully conducted by OPM, VA, and DMA as well as Tupperware and Ford Aerospace.

2.1.2 SIP Components

A SIP is composed of increments, releases, and phases (as illustrated in Figure 2-1). Increments are formed by grouping systems, subsystems or functions to achieve smaller, more manageable subunits. These increments are then moved through a sequence of specific improvements, or releases, which are usually one of three basic types: a conversion release to achieve machine independence, a refinement release to achieve modularity, or an enhancement release to allow for sharing of code modules and resources. Each release is then taken through the four phases of the SIP process: planning and analysis, preparation, improvement, and implementation.

2.1.3 SIP Activities

Figure 2-2 illustrates the activities of a software improvement program that would typically be performed in each release. Note that the functions performed for each release may overlap even though the objectives of each are quite different.

The kinds of activities performed in each of the phases of the software improvement program and the relationship of those phases -- running from the planning and analysis phase (which begins with the inventory and analysis of the applications and systems) to the final acceptance testing and systems transition of the implementation phase -- is illustrated in Figure 2-3.

2.1.4 Software Engineering Technology Support for SIP

An important support mechanism for the SIP is the organization's Software Engineering Technology (SET). An SET consists of five elements which direct and control all software activities throughout the software's life cycle:

- Standards and Guidelines,
- Procedures,
- Tools,

Figure 2-1

A SIP is Composed of Increments, Releases, and Phases

Increments	**Releases**	**Phases**
Logical groupings of systems, subsystems, or functions	Logical groupings of improvements to be performed at one time. Three basic types: Conversion - achieve machine independence Refinement - achieve modularity Enhancement - share code and resources	Groups of activities to define the SIP progression. There are four phases: Planning and Analysis Preparation Improvement Implementation

Underlying Assumptions

- Most major ADP organizations have over a decade of software investment
- Most Federal agencies are highly dependent on ADP systems to perform their missions
- Software maintenance is difficult enough without introducing significant divergence from the existing baseline by conducting application development through major redesigns or new development.

- o Quality Assurance (QA), and
- o Training.

An organization's SET provides the methods, measures, and controls for managing an organization's software activities. The five elements of a SET provide a baseline from which the SIP can operate. Four of the five elements of an SET are actually management, not technical mechanisms. Figure 2-2 illustrates the relationship between the SET and the SIP. As is clear from this diagram, an SIP cannot be developed in isolation from the SET. On the other hand, the most effective way to implement an effective SET would be through an SIP. GSA recommends that the SET and SIP within an organization be developed simultaneously. Based on that recommendation, one of the first steps of the SIP would be to determine the state of the organization's current SET and one of the first logical increments would be those steps necessary to upgrade the SET to support the SIP (e.g., expand or extent standards and procedures, acquire additional support tools.

2.1.5 Criteria for Determining SIP Feasibility

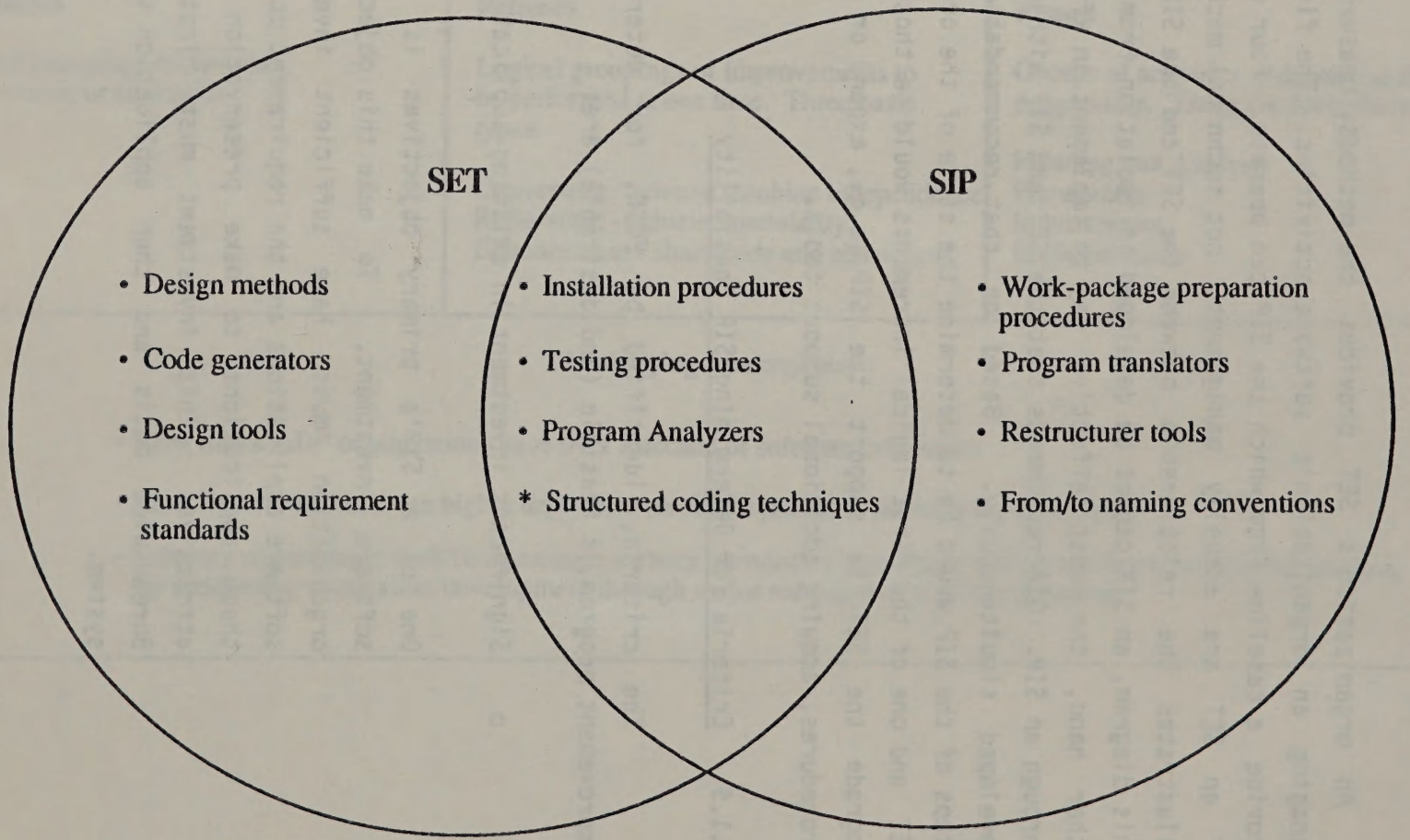
The criteria, identified by FCSC, for determining if a software improvement program is feasible (and desirable) are:

- o Significant investment in existing applications software,

  One of a SIP's primary objectives is to preserve existing software investment. To make this objective worthwhile, the organization must have sufficient investment in existing software applications and the requirement to continue to support these applications to make preservation of that investment attractive. This investment must first be considered on a Bureau-wide basis and then application system by application system.

Figure 2-2

An Organization's SET Provides the Means, Metrics, and Controls an Organization Needs to Properly Execute a SIP.



- Existing software which essentially meets functional needs,

  SIP is a software improvement methodology, not a software development methodology. It is designed to retain the functional integrity of existing software while reducing maintenance costs. This assumes the software is essentially stable and meeting user needs. If the software has serious functional deficiencies, a redesign or rewrite effort would be more appropriate.

- Continuing high investment in software maintenance.

  One major objective of the SIP is to reduce software maintenance cost. This assumes that maintenance cost is high enough that savings from the SIP will more than compensate for the cost and effort of a SIP. High application maintenance cost is difficult to measure quantitatively because it is difficult to establish what the normal maintenance cost should be. However, certain factors or application characteristics make applications more difficult and, hence more costly, to maintain. If applications exhibit these factors, it is reasonable to assume that their software maintenance costs are high.

2.2 GENERAL APPLICATION ASSESSMENT

The General Application Assessment is an overview of the current state of BLM's Bureau-wide applications based on surveys and interviews of key BLM application support staff at DSC and the SOs. The general assessment has two objectives:

- Provide BLM with a reference point on the condition of its applications in terms of maintainability and usability.

- Determine if the state of BLM's applications warranted a more detailed assessment to focus BLM's efforts on specific applications or areas for improvement.

functional deficiencies, a redesign or rewrite effort would be more appropriate.

- Continuing high investment in software maintenance.

  One major objective of the SIP is to reduce software maintenance cost. This assumes that maintenance cost is high enough that savings from the SIP will more than compensate for the cost and effort of a SIP. High application maintenance cost is difficult to measure quantitatively because it is difficult to establish what the normal maintenance cost should be. However, certain factors or application characteristics make applications more difficult and, hence more costly, to maintain. If applications exhibit these factors, it is reasonable to assume that their software maintenance costs are high.

2.2 GENERAL APPLICATION ASSESSMENT

The General Application Assessment is an overview of the current state of BLM's Bureau-wide applications based on surveys and interviews of key BLM application support staff at DSC and the SOs. The general assessment has two objectives:

- Provide BLM with a reference point on the condition of its applications in terms of maintainability and usability.

- Determine if the state of BLM's applications warranted a more detailed assessment to focus BLM's efforts on specific applications or areas for improvement.

Included in this analysis are BLM's Bureau-wide applications systems supported by DSC and those supported by the State Offices.

In this section we will discuss our approach to the General Application Assessment and the two major components of that assessment:

Figure 2-3

A Release is Generally One of Three Basic Types:
Conversion, Refinement, or Enhancement

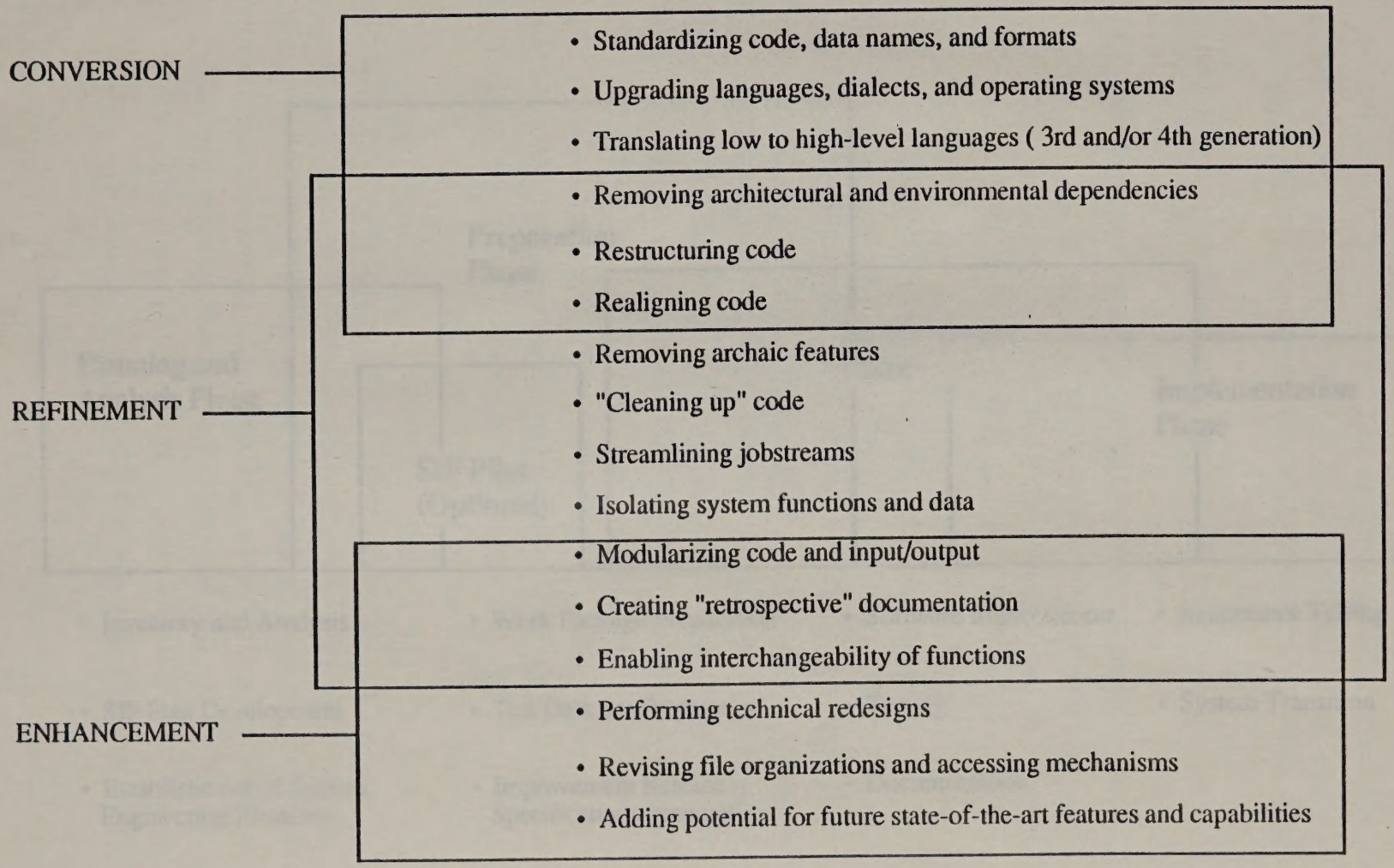
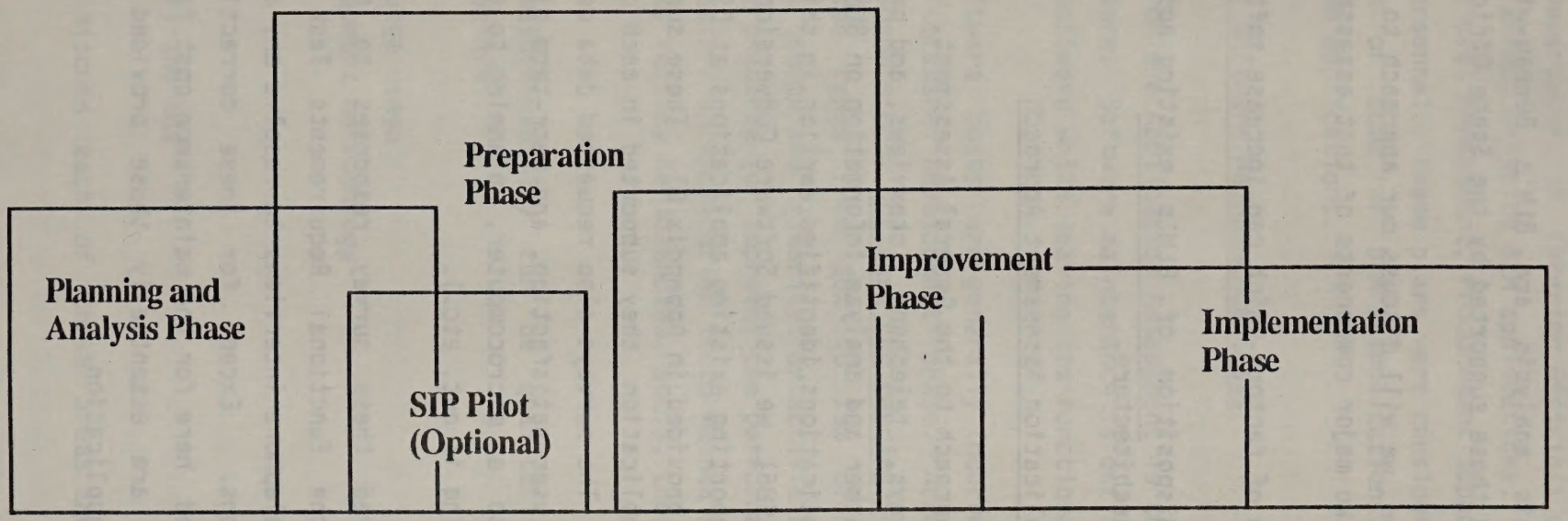


Figure 2-4

The Phases of a Software Improvement Program Run from Planning and Analysis through Implementation



- Inventory and Analysis
- SIP Plan Development
- Establishment of System Engineering Elements
- Work Package Preparation
- Test Data Set Preparation
- Improvement Release Specifications Preparation
- Software Improvement
- Testing
- Documentation
- Acceptance Testing
- System Transition

Included in this analysis are BLM's Bureau-wide applications systems supported by DSC and those supported by the State Offices.

In this section we will discuss our approach to the General Application Assessment and the two major components of that assessment:

- Examination of factors which can increase software maintenance costs
- Probable disposition of BLM's existing applications under the new technical architecture.

2.2.1 General Application Assessment Approach

In our approach to the General Assessment, we used a combination of support staff surveys, telephone interviews, and prior deliverables of the ADEMP project to gather and analyze information on BLM's applications. Based on the list of applications identified earlier in this project (see Existing Capabilities, June 1986), we issued Software Conversion and Assessment Surveys to key BLM staff supporting existing applications at DSC and the State Offices (a sample survey is provided in Appendix B). These surveys requested that the staff rate the application they supported in each of the factors affecting maintenance cost. The surveys also requested data describing change request backlogs, overall user satisfaction, and near-term plans for the application (e.g. moving it to a microcomputer, planning to significantly expand its functionality, phasing it out, etc.).

We then compared these survey responses to the user interviews we conducted during the Functional Requirements Task and to each other. We followed up with telephone interviews to resolve any apparent inconsistencies or misinterpretations. Except for these corrections, the ratings of BLM applications applied here for the maintenance cost factors and the plans for these applications are essentially those provided by the BLM staff most familiar with that application.

We also contacted Department of Interior Officials concerning DOI ADP initiatives which might affect existing BLM applications. These initiatives covered four areas: Financial Management, Procurement Support, Real Property Management, and Payroll/Personnel. These plans were considered in determining the probable disposition of some of BLM's applications, especially BLM's Financial Management System applications.

2.2.2 Factors Affecting Software Maintenance Costs

One major focus of the Application Assessment is the maintainability of the implemented software. Software maintenance is defined as the process of modifying implemented software while leaving its functionality intact.

The majority of software costs are generally incurred during software maintenance, after the software development phase. According to software engineering expert Barry Boehm's book Software Engineering Economics, software maintenance cost estimates range from 50% to 75% of the applications life-cycle cost. These costs are primarily for activities to preserve the existing functionality and preserve (or improve) the performance of a software product. Thus, the easier an application is to maintain (e.g., update to satisfy new requirements, rectify deficiencies, etc.), the lower the cost of supporting it.

An application's maintainability can be qualitatively measured by examining six areas. These are:

- Application language usage
- Design and development methodology
- Documentation
- Data standards
- Test data
- Conversion history and age

The following sections discuss each of these areas and their effects on software maintenance.

2.2.2.1 Applications Language Usage

Overlapping languages can increase the support cost for an application. Each language used in an application should have a distinct role to minimize overlap in functionality and hence reduce overlap in support activities. The language used by an application for a particular role (e.g. FORTRAN for scientific processing and COBOL for business processing) should be the same as those used by other applications in that role. When languages are not given clear and distinct roles, they tend to overlap and produce duplicate modules which essentially perform the same function but are incompatible. This causes higher maintenance and training costs as more than one module performing the same functions has to be supported and complicates application support.

Application code is easier to maintain when it is written in "current" versions of the language, i.e. application code that is no more than one version behind the latest accepted standards (e.g. FORTRAN 77 and FORTRAN 8x). It may also requires less supporting systems software to interface to these languages. When an application is written in obsolete code, the application's support staff must be aware of the nuances and idiosyncrasies of both the older and newer versions. This requires more experienced and hence more expensive staff. Multiple versions of a language require separate compilers for each version.

When multiple versions of the same language are mixed, the resulting applications can sometimes generate unanticipated results. For example, the default values for certain parameters may have changed between versions making the results of computations using these defaults unpredictable or inconsistent in the new version with the results of the exact same code under the old version. The programs themselves may become incompatible. If a subroutine called by a program written in the old code has changed (e.g., the subroutine must now pass or be passed an additional parameter) the results of the new subroutine when called by that program will be unpredictable. Because these problems can occur with no apparent change to the application code, they can be very difficult and expensive to identify and correct. These inconsistencies, or "bugs", can occur whenever language versions change but their

likelihood increases as the number of versions between the obsolete code and the current version increase.

Using one version consistent with the current ANSI standards for the application language (assuming there are ANSI standards for that language) is advisable and reasonable for other reasons than simply keeping the code up-to-date. ANSI releases new standards for a language infrequently -- once every eight to ten years. Updating the code provides an opportunity to take advantage of new coding techniques as well as new language features.

Machine-specific code within applications also tends to increase maintenance and conversion costs. Machine-specific code reduces the portability (the ability to easily move and operate the software in other operating environments) of the software and increases staff training and experience requirements. Machine specific code requires additional resources (time, personnel, etc.) to convert to a new environment. Not only must the programmer must determine comparable commands and features in the new environment are before the application can be converted, but there may also be no comparable technique or feature in the new environment, forcing a costly redesign. Machine specific code also raises the skill required for the support staff and hence the support cost. If an application uses machine specific code, only staff with sufficiently detailed knowledge of the machine can support it.

Vendor-specific languages (such as vendor-specific extensions to ANSI standard languages) should also be avoided. The use of vendor-specific languages limits the portability of applications and raises problems similar to those experienced with machine specific code.

2.2.2.2 Design and Development Methodology

Most application systems developed today are designed and written using structured programming techniques and/or use a fourth generation languages (4GL) to prototype and, where justified, to implement the system. These techniques not only reduce maintenance costs by catching errors early and simplifying the process of error isolation and problem resolution but also

reduce the implementation resources (time and personnel) required. Structured programming methodology requires applications to be written in functional modules of only one or two pages of code apiece. Modularity isolates system functions, allowing modifications and testing to be performed on functions independently. In addition, the design and code should conform to standard structuring and naming conventions (e.g., using uniform formats and variable names) to increase understandability and thereby simplify support.

Another money-saving feature of current development techniques is the "code walkthrough", a detailed review of the application code. Application code should be "walked through" module by module before being system tested to catch errors as early as possible. Errors detected early in the development process are much easier and less expensive to correct than the same errors detected and corrected later during maintenance. Once an application is in production, undetected errors may have already affected the production data or processes and require extensive data recovery as well as correcting errors.

2.2.2.3 Documentation

Well documented systems are much easier to support and maintain. If a system is well documented, the amount of time required for ADP staff support an application is dramatically reduced (e.g., locating a module, tracing where a variable is used). The quality and quantity of available documentation also affects the conversion effort. Good documentation enables the conversion staff to gain a faster understanding of the application. It also helps determine whether or not the converted application is operating correctly during testing (e.g., the functionality, as defined in the original documentation, is still intact).

The documentation for applications should be complete and substantially up to date. It should include functional requirements, systems specifications, program specifications, jobstream documentation, operational instructions, restart recovery procedures, and user documentation.

2.2.2.4 Data Standards

Data standards can reduce maintenance effort and cost. Data standards provide for the consistent and uniform use of variable names or symbols across applications increases the understandability of software programs and facilitates building interfaces between applications. With a clear understanding of how and where data is used, program corrections and modifications are easier to perform and test.

To keep data maintenance costs down, the data within the Bureau-wide applications should be coordinated through a central data element dictionary which controls form, content, composition, precision, accuracy, and access for each data element. It should also include narrative as to the nature of the data element and who is responsible for creating, updating, and using the data element. These data elements should be named in accordance with standard data naming conventions as established in the organization.

2.2.2.5 Test Data

Test data reduces overall maintenance costs by making testing of applications easier and more comprehensive. The ease with which software changes can be demonstrated to be complete and correct is an intricate part of the maintainability of an application. Problems may arise at a later date if certain untested "cases" generate inaccurate results. If a program's logic is thoroughly tested upon implementing a software modification, incorrect results or effects on other functions performed in the application can be identified and directly traced to the particular change, thus eliminating a major search for the point of error and possible contamination of production data.

Developing good test data is difficult and costly. The programmer must consider many conditions both singly and in combinations, including unexpected combinations. According to GSA's Federal Software Management and Conversion Support Center, test data should exercise a minimum of 70% of the program logic. In addition to the test data itself, a library of benchmark results should be maintained to illustrate the results of the software when it is operated correctly. As applications are modified, additional test data

records would be added to the test data and the new results placed in the benchmarks results library. This provides for more complete and consistent tests of the applications as they are changed. It is also absolutely necessary to confirm that the software has been successfully converted to run on another manufacturer's equipment.

2.2.2.6 Conversion History and Age

An application's conversion history and age affects the cost of support. Older applications (those developed before modern programming techniques were implemented) and applications which have been converted are often more difficult to support and hence more expensive. Older applications were generally developed with different objectives than ease of support (e.g., minimize memory requirements). Standards of "good programming" have also changed. Applications that have been converted are often converted directly (syntax changes only) with as little code rewrite as possible. As a result, the code executing a particular operation may not be written in such a way as to make optimal use of the new machine.

2.2.3 Application Disposition

We separated BLM's Bureau-wide applications into five classes depending on their probable disposition (i.e. BLM plans for these applications). We based these probable dispositions on the results of the Software Conversion and Assessment Surveys completed by BLM staff. The disposition of an application affects whether additional resources should be invested in it to improve it or extend its life and how much investment is warranted. We then considered a range of six improvement options from archiving (i.e. no improvement) to complete redevelopment based on the characteristics of each application class.

The five classifications are as follows:

- Discontinued or inactive applications

  These applications were identified by BLM in the original list of DSC applications (provided to AMS by DSC ADP staff) but will not run on the new system. They include applications which have already been archived, applications which are currently being implemented on microcomputers, and applications which will have a counterpart on the new system, such as the charge-back accounting system which would be replaced by the new computer's own accounting system.

- Superseded by DOI applications

  This class contains those applications which are likely to be substantially superseded by DOI-wide applications. The Department of Interior has targeted four areas for DOI-wide systems: payroll/personnel, procurement support, financial management, and real property management.

- Superseded by BLM applications

  This class contains applications which would be replaced by other applications currently under development within BLM. In particular, the Automated Lands and Minerals Recordation System (ALMRS), the Automated Resources Data (ARD), and the Geocoordinate Data Base (GCDB) efforts are expected to replace several existing systems.

- Continuing applications-large

  This class includes applications which will not be discontinued or superseded in the near future and exceed five thousand lines of code. The median size of BLM's continuing applications is approximately five thousand lines of code, with roughly the same number of applications over five

thousand lines as under. Thus, we used five thousand lines of code to separate "large" and "small" applications.

Large applications usually represent a higher software investment and so warrant more attention. In conventional inventory management, it is common to separate high-investment components from lower-investment components and concentrate management attention on the high-investment components. In systems development, investment is often measured in terms of lines of code. Following this practice we have separated those applications in which BLM has made a high investment (i.e., five thousand lines of code or more) from those in which BLM has made a relatively low investment. While there are many measures of the relative importance of applications (e.g., mission criticality, volatility, leverage -- these are discussed in Chapter 6), relative investment is one of the few that can be considered objectively.

- Continuing applications-small

  This class includes applications which will not be discontinued or superseded in the near future and are less than five thousand total lines of code.

Once we classified BLM's applications we identified six disposition alternatives BLM might possibly take for each of the applications in each of the application classifications were identified. These alternatives are:

- Archive - Store the application on tape in an archive without ever converting to the new technical environment.

- Convert - Modify the application as necessary to have it operate in the new technical environment in essentially the same fashion as it did in the old environment (i.e. no increase in functionality).

- Enhance - Provide modest or isolated improvements in functionality but leave the vast majority of the application untouched other than what is necessary to convert it to the new environment.

- Rewrite - Modify the application code to make it more efficient or easier to support in the new technical environment but without changing the overall application design or the functional requirements the application serves.

- Redesign - Reconsider the way the application is meeting the functional requirements and design the application to take full advantage of the new technical environment. The functional requirements remain unchanged. This implies a subsequent rewrite.

- Redevelop - Revisit the application's user requirements to see if additional functionality is required to meet changing user needs. This implies a subsequent redesign and rewrite to meet the new requirements.

These actions start progressively farther back in the life cycle management process, ranging from "archive" and "convert", which do not require reentering the life cycle management process, to "redevelopment" which requires starting at the very beginning of the life cycle management process and revisiting the functional requirements.

2.3 DETAILED APPLICATION ASSESSMENT

Upon completing the general assessment of BLM Bureau-wide applications, we conducted a detailed application assessment for those applications which we classified as large and which BLM expects to operate (either wholly or partially) in the new technical environment. These applications represent a significant investment by BLM and through that an implied importance to BLM's mission. The results of the detailed assessment for each application is included in Appendix C.

We interviewed the support staff for each continuing large application for more detail with respect to the factors affecting maintenance costs (described in the general assessment). In addition, we used a computer model called the Constructive Cost Model (COCOMO) to estimate the resources necessary for developing these applications from scratch in order to provide BLM with upper cost bounds on the amount of resources BLM should be willing to invest in improving (versus redeveloping) these applications. A summary of the COCOMO results and a more detailed explanation of COCOMO are included in Appendix D.

We selected the COCOMO model for estimating redevelopment costs for two reasons. The first reason is that the model is easy to understand. The developer of the model, Barry Boehm, a widely recognized software engineering expert, has detailed the development and justification for COCOMO in his Software Engineering Economics textbook. The second reason is that the model is fairly accurate. According to Boehm, the intermediate version of the model (the version we applied) is within 20% of the total cost of actual efforts 68% of the time in a sample of 63 large application development projects which is comparable or better than other estimating approaches. We describe the COCOMO model more fully in the next section.

2.4 COCOMO COST ESTIMATE MODEL

COCOMO, was developed at TRW by Barry Boehm. Published in 1981, the model estimates application development staff cost and effort based on estimated lines of code, project "mode" (an estimate of project difficulty), and fifteen "cost drivers" that affect productivity.

COCOMO has been used extensively within AMS and by other private industry firms to estimate software development costs. In this analysis, we used the model to set an upper bound for the resources BLM should be willing to invest in enhancing their existing software. With the projected cost to redevelop the application from scratch estimated, the cost the Bureau should be willing to absorb to enhance the application should be somewhere below that. A full

description of the development and justification of the COCOMO model can be found in Boehm's book, Software Engineering Economics[1].

2.4.1 COCOMO Cost Drivers

The fifteen cost drivers in the COCOMO model are attributes of the end products, the computer used, the personnel staffing, and the project environment.

The complete list of factors (and their abbreviations which appear on the spreadsheets included in Appendix B) are as follows:

- Product Attributes
  - Required Software Reliability (RELY)
  - Data Base Size (DATA)
  - Product Complexity (CPLX)
- Computer Attributes
  - Execution Time Constraints (TIME)
  - Main Storage Constraints (STOR)
  - Virtual Machine Volatility (VIRT)
  - Computer Turnaround Time (TURN)
- Personnel Attributes
  - Analyst Capability (ACAP)
  - Applications Experience (AEXP)

---

[1]Boehm, Barry W. Software Engineering Economics, Prentice-Hall Inc., Englewood Cliffs, N.J., 1981.

- Programmer Capability (PCAP)
- Virtual Machine Experience (VEXP)
- Programming Language Experience (LEXP)

o Project Attributes

- Modern Programming Practices (MODP)
- Use of Software Tools (TOOL)
- Required Development Schedule (SCED)

The values, which range from extremely low (one) to extremely high (five), given to each of these cost driver attributes for each application examined were based on phone interviews with BLM support staff. Together they determine a multiplying factor which is applied to estimated lines of code to estimate the software development effort.

2.4.2 COCOMO Assumptions

A few assumptions underlying the use of COCOMO are key to our results and are identified below:

o COCOMO assumes the software development process is composed of ten phases:

1 - Feasibility
2 - Organization and Planning
3 - Software Requirements Specification
4 - Product Design Specifications
5 - Detailed Design Specifications
6 - Code
7 - Unit Test
8 - Integration and Test
9 - Acceptance Test
10- Operation and Maintenance

- The development period covered by the model's cost estimate includes the product design phase (4) through the end of the integration and test phase (8).

- The requirements specification will remain substantially the same after the requirements phase has ended. Some refinements and reinterpretations are assumed, but any significant modifications should be covered by running a revised cost estimate.

- AMS assumed average ratings for the personnel attribute factors to avoid tying estimates too closely to current staff capabilities. This provides a slightly higher but more generic cost estimate.

2.5 DATA SOURCES

We collected the data used in this analysis through the following sources:

- BLM staff using Software Conversion and Assessment surveys, phone interviews, and working sessions.

- BLM guidance documents and handbooks

- Past deliverables of the ADEMP project in particular the Functional Requirements (Task 1, April 1986) and Existing Capabilities (Task 2, June 1986).

- Federal and industry publications

Appendix E provides a complete list of published references.

3. GENERAL APPLICATION ASSESSMENT

This chapter contains the results of our initial steps in the process of assessing the feasibility of implementing a SIP. These initial steps consisted of:

- Reviewing BLM's 58 Bureau-wide application systems in terms of the six factors affecting software maintenance costs described in Chapter 2: use of application languages, design and development methodology, documentation, data standards, test data, and conversion history and age.

- Identifying potential disposition alternatives for these systems:archive, convert, enhance, rewrite, redesign, or redevelop the application.

- Suggesting disposition actions for BLMs consideration for the different categories of systems.

Generally, our assessment indicates the more recently developed applications such as the Automated Fleet Management and Wildlife Information systems employ newer and more efficient development and maintenance methodologies (e.g. structured design and programming, etc.), standards (use of specific subroutines, documentation and comments in the programs, etc.), and tools (fourth generation languages, etc.) resulting in more efficient software and therefore reduced maintenance costs. The "older" BLM application systems such as the Waterpower and Cadastral Survey Field Note Systems, developed in the 1970's, use first generation software development methodologies and tools. As a result, maintenance of these older systems is more costly and difficult.

44 applications were identified as being partially or wholly implemented on the new architecture. Fourteen applications were also identified as being discontinued or inactive. These fourteen applications are either being

converted for implementation on minicomputers or have a counterpart function in new application systems such as ALMRS.

The overall result of this initial assessment process indicated BLMs applications meet the general profile for implementing a SIP.

3.1 ASSESSMENT OF MAINTENANCE FACTORS

The first step in the assessment of BLM's 58 Bureau-wide applications was to review them with respect to the six major factors affecting maintenance costs as defined in Chapter 2. As shown in Figure 3-1, our findings indicate that BLM probably incurs high application software maintenance costs due to the state of this software with respect to these factors. The results of our review supporting this assessment are presented below.

3.1.1 Use of Application Languages

Overall, BLM has developed standards and guidelines and enforces them in the application system development and maintenance process. As described in Chapter 2, these standards and guidelines should ensure that current ANSI standard versions of languages are used, each language used has a definite role, and the language itself does not complicate conversion (i.e. no machine-dependent code, vendor specific languages, or obsolete versions of a language).

Over 90% of the Bureau-wide application code supported by BLM is written in either COBOL or FORTRAN. Most of this code is within ANSI standards. BLM uses COBOL and FORTRAN in specific roles (e.g., COBOL for business applications and FORTRAN for scientific and computation-intense applications). BLM has limited vendor-specific languages to specific roles although these roles sometimes appear to overlap (e.g., the use of ASPEN/2 and INFO).

BLM, however, uses a large number (eight) of languages (COBOL, FORTRAN, ASPEN/2, BASIC, DEF II, TEX, INFO, and DM-IV) as well as both obsolete and more current versions of the same language (e.g. COBOL 68 and COBOL 74; FORTRAN 66 and FORTRAN 77). Approximately 14% of BLM's COBOL is written in

Figure 3-1

Results of the General Software Assessment Indicate High Maintenance Costs for Many BLM Applications

Application Languages • Large number of languages (8) • Obsolete versions (e.g. FORTRAN 66) • Hardcoded machine-specific parameters (e.g. screen processing) • Vendor-specific languages (e.g. ASPEN)	**Design and Development Methodology** • Currently using structured techniques • Many applications written prior to implementation of these techniques	**Documentation** • BLM survey responses rated documentation 4.3 on a ten point scale. • No correlation between documentation quality and system size or test data. • Newer systems appear to have better documentation
Data Standards • Data Element Dictionary is a home-grown data dictionary • Data naming conventions are used • DED used by many applications (not all) and is required by the DSC Computer Applications Handbook • Lack of bureau-wide DBA function	**Test Data** • 40% of logic paths tested by existing test data as compared to 70% FSMSC recommends • No requirement or allowance in current procedures for generation and maintenance of test data libraries and benchmark results libraries.	**Conversion History and Age** • Many old applications (14% of the COBOL programs are COBOL 68) • Some applications were converted before in the late '70's.

COBOL 68. This complicates support. Staff must know both current and past versions of the language used. Using many languages reduces staff mobility and increases training requirements by requiring a more extensive and application-specific set of skills. To address this situation, WO-770 is in the process of issuing a workplan directive to DSC for fiscal '88 to upgrade the COBOL 68 programs to COBOL 74 at DSC.

Five of the eight languages supported by BLM are vendor-specific (ASPEN/2, DEF II, TEX, INFO, and DM-IV). Using vendor-specific languages increases conversion and training costs since neither the software nor the training can be applied to a new, dissimilar technical environment. It is also unavoidable. Certain functions, such as screen processing, can only be performed by using machine specific commands and language extensions. BLM has successfully limited the amount of code using vendor-specific languages (less than 10%). No standards have been developed or applied to the code using these languages.

BLM also has device-specific code embedded in some of its older applications (e.g., such as code which interface with a variety of specific terminals). The application code must be modified and recompiled whenever a new device is included. This complicates both including new devices and converting the applications to a new technical environment.

As previously stated, BLM controls language use in its Bureau-wide applications. However, to improve control and application supportability, we recommend BLM:

- Define the roles of those languages other than COBOL and FORTRAN which are used in Bureau-wide applications;

- Define the roles and guidelines for the use of a fourth generation language (4GL) for prototyping and implementation;

- Develop standards and guidelines for using these other languages and for the use of their successors in the new technical environment;

- Reduce the number of languages supported to the fewest number practical, based on the roles defined for each language; and

- Upgrade obsolete versions of current languages to the versions the future environment will support.

3.1.2 Design and Development Methodology

BLM's Bureau-wide applications are split along historical lines with respect to the use of current design and development methodologies. The more recent applications use structured design and development techniques, code walkthroughs, and quality assurance mechanisms. They are modular and conform to the coding and naming-convention standards specified by DSC's Division of Computer Applications (D-220) in the Computer Applications Handbook (H-1262-2). This Handbook outlines the standards and procedures ADP staff are supposed to follow when developing Automated Data Systems (ADS).

The older applications (five years or older) were not developed using these techniques. They are not very modular nor do they adhere to any particular coding standards. As is often found in older applications, some applications contain logic which branches around "dead" sections of code which are never executed. As a result, they are difficult to maintain, and do not make efficient use of CPU and memory resources.

3.1.3 Documentation

Overall, existing BLM applications are not well documented. Efforts are currently underway, however, to improve this situation. The BLM ADP staff rated existing documentation on a scale of 0 to 10, where 0 corresponded to "No Documentation" and 10 corresponded to "Complete and Up-to-Date Documentation". As shown in Figure 3-2, the average, for the 44 applications examined (14 application are to be discontinued or inactive) was 6.6. 14% of the applications examined have complete documentation. 41% were rated as having insufficient documentation (little, unusable, or no documentation at all).

Figure 3-2

14% of BLM's Currently Active Applications Were Rated by BLM ADP Staff as Having Complete Documentation

Value	Description	# Applications
10	Complete and up-to-date	1
9	Complete but somewhat out-of-date	5
8	Extensive amount which is incomplete but useable and up-to-date	3
7	Extensive amount which is incomplete and out-of-date but useable	2
6	Extensive amount which is incomplete, not useable, and out-of-date	0
5	Moderate amount exists which is incomplete but useable and up-to-date	8
4	Moderate amount which is incomplete and out-of-date but useable	7
3	Moderate amount but incomplete, not useable, and out-of-date	4
2	Very little exists but is up-to-date	5
1	Very little exists and is out-of-date	6
0	No documentation	3
	TOTAL	44

BLM's newer applications are well documented (i.e., complete and up-to-date operations, user, system, and program documentation conforming to BLM standards and guidelines). The older applications do not have this level of documentation. As with application development methodology, the DSC Computer Applications Handbook outlines the documentation to be prepared during the ADS development process. BLM's newer application systems conform to these guidelines. Without well documented systems, the Bureau has difficulty in such things as reviewing the program logic, changing/conveying user interface procedures, maintaining data file structures, performing back up and recovery procedures, etc. As a result, the ADP support staff has a difficult time maintaining application software and, in addition, software conversion to a new technical architecture will be fairly more difficult.

However, in accordance with the Computer Applications Handbook guidelines, application systems that are currently being rewritten or modified (e.g., the Automated Fleet Management System which will replace the Motor Vehicle System) are reported as being thoroughly documented as the system development progresses.

3.1.4 Data Standards

The Computer Applications Handbook outlines standard naming conventions and formats to be used in writing COBOL and FORTRAN programs. The Bureau also maintains a Data Element Dictionary (DED), which is composed of data elements and other information.

The newer applications generally use standard naming conventions and the DED. Most of the older applications, however, were written prior to the publication of the Handbook and do not conform to current BLM data standards. Specifically, they do not make use of the DED nor do they use standard naming conventions. Some of the older applications do use consistent naming conventions across the programs comprising them.

The lack of data standards and limited use of the DED by the older applications causes several problems for BLM, including increasing ADP support for application maintenance (due to the increase in programmer time to

familiarize themselves with the data elements in an application and maintain them), the inability to integrate applications, and the generation of inaccurate or incomplete information (due to redundant data files and inconsistent data definition).

3.1.5 Test Data

Current BLM ADS procedures require that a small test file of valid and invalid records be created to test the program logic (to ensure it produces accurate results).

Of the 44 BLM applications examined, test data is reported to exist for only 20 applications. Of these 20, 13 were rated as having test data that exercises 100% of the application's program logic. 24, or 55%, of the applications were reported as having no test data at all. This has two effects:

- Test data must be developed from scratch whenever a change is made, and
- BLM's conversion effort will be increased as BLM must also develop test data to verify that the converted applications perform accurately.

3.1.6 Conversion History and Age

Applications written prior to the installation of BLM's Honeywell 66/80 (March, 1978) were run on a Burroughs computer system (e.g., the Lease Management System). Because they were created prior to the establishment of standard ADS development procedures, these, and other older applications, do not conform to the standards (i.e., naming conventions, DED, program format, documentation requirements) contained in BLMs Computer Applications Handbook. This makes the applications more difficult to support. It is also likely the code in these applications is not optimizing the use of the Honeywell system.

3.2 APPLICATION CLASSIFICATIONS

As discussed in Chapter 2, we separated BLMs 58 Bureau-wide applications into five classes depending upon their probable disposition. Figure 3-3 shows that 35 (approximately 60%) of the 58 Bureau-wide applications running on the Honeywell DPS-8/70 will be either discontinued or superceded by ALMRS, DOI planned initiatives, or are currently inactive. As also indicated in Figure 3-3:

- Twenty-one applications will be partially or wholly superceded by ALMRS or DOI's Financial Integrated Review for Management (FIRM). Many, if not all, of these applications may be initially implemented on the new ADEMP system until the new, replacement applications become operational. These 21 applications may therefore be converted to the new ADEMP system. Enhancements to these applications, however, is not currently warranted.

- BLM expects fourteen applications to be discontinued -- either because they are no longer going to be used, they are being converted to run on microcomputers, or they have a counterpart function in new applications such as ALMRS.

- Of the twenty-three applications to be fully implemented in the new technical environment, eleven are considered small in terms of the number of lines of code (less than 5000 lines).

3.3 ALTERNATIVE ACTIONS FOR EACH CLASSIFICATION

There are six alternative disposition actions that may be taken for these 58 application systems. Figure 3-4 provides a table of the viable actions for each application category. Each alternative action corresponds to a phase in the software life cycle. For example, "redevelopment" corresponds to the development phase of the software life cycle in which the functional requirements are defined, while redesign assumes the same functional

Figure 3-3

Many of BLM's Bureau-wide Applications will be Run in the New Technical Environment

Discontinued/inactive/replaced
ACLDS
AFILMS
AIRS (MS-1)*
Chargeback****
Data Element Dictionary****
Digital Elevation Model*
ESO Minerals*
ESO Patent Index*
General Entry Trans. Proc.
Health Risk Appraisal
Payroll costs
Procurement Planning
SLMS*
Suspense Accounts

Superceded by BLM applications
Bonds Subsystem
Budget Matrix
Case Recordation
Master Name
Mining Claims

Superceded by DOI applications
Aging of Accts. Rec.
Automated Fleet Mgmt.**
Deposits in Transit
Financial Mgmt. Edits**
Financial Mgmt. Repts.**
Financial Mgmt. Year-end**
Fire Cost Reports
General Ledger
Imprest Fund
Lease Management**
Payroll/Personnel
Perpetual Inventory
Personal Property Mgmt.**
Reimburseable Billing**
RMAS***
Travel Advance

Continuing Applications - Small
Aviation Contract
Emergency Fire Fighters
Forest Models
Forest Utilities
Job Documentation Rept.
Mgmt. By Objectives
Planning Schedule
Public Domain Forecast
SAGERAM
SIMO
Water Data

Continuing Applications - Large
Adopt-A-Horse
Cadastral Field Notes
Checks to Treasury
Inventory Data
Material Sales*
Operating Budget
Program Management
Summer Hire System
USFS Forest Inventory
Waterpower System
Wildfire Reporting
Wildlife Information

* Part superceded by ALMRS, part to be implemented on micros
** Part superceded by DOI, part to continue
*** Part superceded by DOI, part to be implemented on Micros
**** Purchased with the replacement system/to be replaced

Figure 3-4

Different Alternative Actions are Viable for Applications in Each Disposition Category

Legend: ● Yes ◍ Yes, if appropriate ○ No

Alternative Actions / Application Type	Archive	Convert	Enhance	Rewrite	Redesign	Redevelop
Discontinued or inactive	●	○	○	○	○	○
Superceded by BLM applications	○	◍	○	○	○	○
Superceded by DOI applications	○	●	◍	○	○	○
Continuing Applications -Small	○	◍	●	◍	◍	◍
Continuing Applications -Large	○	◍	●	◍	◍	◍

requirements but provides for a new technical solution for executing them. In all alternatives save archiving, a SIP may be used. These potential dispositions are discussed in the following sections.

3.3.1 Alternative 1: Archive The Application

Applications which are no longer used, or will not be used when the new technical architecture is implemented, should be archived (i.e. the programs, job control deck, and any specific data files are removed from the system and stored on magnetic media, usually tape, so that if needed in the future they can be restored onto the system for execution). There is no need to transfer these applications to the new technical environment unless they must be implemented on the new architecture for some interim period of time. All discontinued or inactive applications fall into this category.

3.3.2 Alternative 2: Convert The Application

Application conversion entails transforming the software, without any functional modifications, so it may be used in another technical environment. Conversion in this context is largely a straight-code conversion. Only those modifications to the code are made which are absolutely necessary to implement it into the constraints of the target environment (e.g., the existing environment and architectural dependencies are removed from the software code, code is standardized, etc.). Improvements are minimal, as most of the conversion effort is concentrated on making the converted software operational in the new technical environment.

Conversion must be performed on applications being moved to a dissimilar computing environment. This alternative is potentially appropriate for all applications (with the exception of those discontinued). It is most appropriate for those applications which rate high in all six major factors affecting maintenance costs. This is the current appropriate alternative for those applications identified as being totally replaced.

3.3.3 Alternative 3: Enhance The Application

Application enhancement involves improving the functionality or performance of an application. This can be done in conjunction with conversion or performed after the software has been converted. Enhancement includes activities such as the modernization of the software code to a state-of-the-art status (i.e., latest version of programming languages, operating systems, etc.) and redesign and redevelopment of small portions (i.e., less than 25%) of existing code to optimize the software in the new environment.

This alternative may be appropriate for those small and large applications which are expected to be operational in the new technical environment. Enhancement is not a viable action for those applications that are expected to be discontinued (for obvious reasons) or those that will be superceded by BLM applications which are currently under development. These enhancements should be completed by the time the new technical architecture is fully implemented.

3.3.4 Alternative 4: Rewrite The Application

Rewriting an application system should be considered for those systems which functionally meet the users requirements but require modification of large portions of existing code (greater than 25%) to enhance its efficiency in terms of the use of the programming languages, data standards, test data, and documentation. The activities for a rewrite are the same as those for enhancement, but on a larger scale. The basic design of the application remains the same.

Those small and large application systems which are not expected to be superceded by other initiatives in the new technical environment are potential candidates for rewrite if a significant amount of code is to be modified.

3.3.5 Alternative 5: Redesign The Application

Redesigning an application system entails developing a new solution to better accomplish the same functions being performed by an existing

application. Those applications which will significantly benefit from a new technical solution are candidates for redesign. For example, an application may be redesigned to use an array processor for its complex numerical operations so that it executes faster or redesigned to provide significant interaction with the users for data input, query and reporting capabilities.

It may be appropriate to redesign some of the small and large applications which are expected to be operational in the new technical environment.

3.3.6 Alternative 6: Redevelop The Application

Redevelopment of an application involves starting at the beginning of the development phase of the Life Cycle Management process. This alternative should be considered when the current application does not meet the functional needs of the users. This action may be appropriate for an application which is expected to be operational in the new technical environment if it is determined that the application is significantly functionally deficient.

3.4 CONCLUSIONS AND NEXT STEP(S)

As shown in Figure 3-3, of the 23 application systems designated to be fully implemented and continuously operational on the new ADEMP system (as opposed to implemented and operational until superceded), 12 are considered large in terms of lines of code. In addition to these 12 systems, 7 of the applications being partially superceded by the DOI FIRM initiative, are also currently large. The amount of functionality to be replaced by the FIRM initiatives within these 7 systems is not currently defined. As a result, for planning purposes, the remaining portions of these 7 systems to be maintained within BLM are also considered in the analysis of the large BLM application systems to be implemented on the new ADEMP system. Figure 3-5 contains a list of these nineteen large application systems.

In considering the sequencing of the SIP efforts, other factors such as mission criticality, system stability/volatility (meeting user requirements without significant continuous maintenance/enhancement efforts) and efficient

EXHIBIT 3-5
Nineteen Bureau-Wide Applications were Examined with Respect to Maintainability

HB	Adopt a Horse
AFMS	Automated Fleet Management*
CS	Cadastral Survey Field
FC	Checks to Treasury
FY	Financial Management Year End Reporting*
FE	Financial Management Edits*
FR	Financial Management Reports*
ID	Inventory Data System
LM	Lease Management*
MS	Material Sales
OB	Operating Budget
PG	Program Management
APPS	Property Management*
FJ	Reimbursable Billing*
SH	Summer Hire System
PF	USFS Forest Inventory
WP	Waterpower System
AFFM	Wildfire Reporting
WL	Wildlife Information System

* These applications will be partially subsumed by DOI's FIRM

use of computer resources (e.g. CPU, disk, etc.), projected functionality improvements (e.g. interactive input/output, query, etc.), etc. must also be considered.

Chapter 4 contains a further discussion of the maintenance status, planned improvements, and projected (high-side) redevelopment costs as generated by the COCOMO Model for the nineteen large application systems.

4. DETAILED APPLICATION ASSESSMENT

This chapter provides a detailed examination of the nineteen, large Bureau-wide application systems which would be the core of the initial focus of a SIP. Included in these nineteen large application systems are those which will be partially subsumed by DOI's Financial Integrated Review for Management. This examination should be used to further refine BLM's disposition actions for these nineteen systems and assist in the development of a detailed SIP plan.

The first section of this chapter presents our in-depth, specific, review of these nineteen application systems, using, as a framework, the key factors which drive maintenance costs. The second section reviews these applications in terms of meeting users requirements, lists the currently scheduled enhancements, and provides estimated costs for redeveloping these applications.

As summarized in Figure 4-1, 13 of the 19 systems are less than 10 years old, 10 do not use vendor specific languages, 14 use naming conventions, 11 use data element dictionaries, and 9 have test data. Appendix C contains the detailed data for each application examined.In addition to data on the six maintenance factors, Appendix C also contains additional data on the application -- such as software reliability required, complexity of the application code, etc., which was required as input to the COCOMO cost model. This data was obtained through telephone interviews with BLM support staff at the DSC and State Offices. The costs for development of each application, estimated by the COCOMO model, are also included.

The objective of this costing exercise using the COCOMO model is to provide another measure to BLM to be used to determine the disposition of these application systems (e.g. redesign, enhance, convert, etc.). AMS is not recommending the redesign of these systems. The cost for the total redesign of all nineteen application systems is projected by the COCOMO model to be approximately $4,566,300. This cost estimate, however, should only be considered as an upper limit when comparing the costs to convert, modify, or

FIGURE 4-1
Newer Applications Conform to BLM's Standards for Computer Applications

Application Name	Age	Languages	Overall Documentation	Naming Conventions	Data Element Dictionary	Test Data	Logic Tested	Converted
Adopt-a-Horse	1 yr	ASPEN/2,FORTRAN	3	no	no	no		no
Automated Fleet Management	3 mos	COBOL, DM-IV	7	yes	yes	yes	100	no - replacing older
Automated Personal Property System	2 yrs	COBOL	4	yes	yes	yes	100	yes
Cadastral Field	pre-'72	ASPEN/2, COBOL	3	yes	some	no		yes - straight code
Checks to Treasury	<1 yr	COBOL	4	yes	no	yes	95	no
Financial Management Edits	15 yrs	COBOL	0	newer programs	newer programs	no		older programs
Financial Management Reports	15 yrs	COBOL	4	newer programs	newer programs	no		older programs
Financial Management Year-End Reporting	15 yrs	COBOL	1	newer programs	newer programs	no		older programs
Inventory Data	2 yrs	ASPEN/2, COBOL	2	yes	yes	yes	100	vegetation data
Lease Management	13 yrs	COBOL	0	yes	yes	yes	90	yes
Material Sales	3-10 yrs	BASIC, COBOL, CRUNS, DM-IV, TEX	1			no		no
Operating Budget	3 yrs	DEF II, COBOL	4	yes	no	no		no
Program Management	2 yrs	COBOL	8	yes	some	no		no
Reimbursable Billing	4 yrs	COBOL	7	yes	yes	no		no
Summer Hire System	6 yrs	COBOL	9	no	no	yes	85	no
USFS Forest Inventory	5-7 yrs	FORTRAN IV, TEX	3	no	no	yes	100	no
Waterpower System	5 yrs	CRUNS, FORTRAN, TEX	4	yes	no	yes	60	no
Wildfire Reporting		COBOL, TEX	5	no	yes	yes	100	no
Wildlife Information System	3-6 yrs	ASPEN/2, COBOL, TEX	5	yes	yes	no		no

redevelop an application system. It should also not be directly compared to the total conversion costs contained in the Conversion Study, Deliverable 6/7.3. The conversion costs, as dictated by GSA, include costs for items such as training, site preparation, etc., which are not part of the COCOMO Model.

4.1 THE CONDITION OF CONTINUING LARGE APPLICATIONS'

This section describes the current condition of the nineteen, large BLM applications which are expected to be transferred to the new technical environment in terms of the six maintenance cost factors we have previously described. Since software maintenance costs range from 50% to 75% of the overall software lifecycle costs, maintainability is an important factor to consider when determining what to do about existing software (i.e., modify, rewrite, or leave as is).

4.1.1 Use of Application Languages

All of the applications are written primarily within ANSI standards (for COBOL, FORTRAN, and BASIC). As shown in Figure 4-2, more than half of these application systems also use vendor-specific languages (e.g., DM-IV, ASPEN/2, TEX, DEF-II, and CRUNS). As a result, those portions of these systems written in a vendor-specific language will have to be rewritten rather than converted when moving to the new ADEMP system.

As detailed in Figures 2-5 and 2-12 of the Software Conversion Study, Deliverable 6/7.3, approximately 6% of the DSC's application systems (94,000 lines of code) and 1% of the State Office application systems (13,500 lines of code) are written in vendor-specific languages. (Note: In addition to those portions of the application systems' code using vendor-specific languages, there is a significant amount of complex code - 26% or 745,000 lines of code - which has been identified as requiring major program logic modification in both the DSC and the State Offices.)

Figure 4-2

Eight Different Application Languages are Used

Application	Languages
Adopt-A-Horse	ASPEN/2, COBOL
Automated Fleet Management	COBOL, DM-IV
Automated Personal Property System	COBOL
Cadastral Survey Field Note System	ASPEN/2, COBOL
Checks to Treasury	COBOL
Financial Management Year-End Reports	COBOL
Financial Management Edits	COBOL
Financial Management Reports	COBOL
Inventory Data System	ASPEN/2, COBOL
Lease Management	COBOL
Material Sales	BASIC, COBOL, CRUNS DM-IV, TEX
Operating Budget	DEF II, COBOL
Program Management	COBOL
Reimbursable Billing	COBOL
Summer Hire System	COBOL
USFS Forest Inventory	FORTRAN IV, TEX
Waterpower System	CRUNS, FORTRAN, TEX
Wildfire Reporting	COBOL, TEX
Wildlife Information System	ASPEN/2, COBOL, TEX

4.1.2 Design and Development Methodology

More than half of the applications as reported by BLM support staff are structured and modular. The more recent applications were designed and developed using structured coding methodologies and conform to the standards of BLM's Computer Applications Handbook. Some of the older applications are hybrids, with old and new parts. These applications (e.g., Financial Management Reports, Financial Management Year-End reports, and Financial Management Edits) were originally written without structured coding, modularity, and naming conventions, but have since been expanded, and the newer programs do conform to BLM's standards for ADS design and development. Figure 4-3 contains additional details on the design and development methodologies for these systems.

4.1.3 Documentation

The average rating given to the documentation available for the nineteen applications, as detailed in Figure 4-4, is 3.4 out of a possible 10. Because the ratings applied to documentation that exists (and not documentation under development), this figure understates the condition of application documentation. Three of the applications which received low ratings (i.e., Checks to Treasury, Inventory Data Systems, and Lease Management) are in the process of being documented. In addition, as the COBOL 68 programs are converted to COBOL 74, the system's documentation will be developed.

Many of the applications are inconsistent in their level of documentation. They have good documentation in certain areas and weak, or nonexistent, documentation in others. For example, the program and user documentation for the Summer Hire application was rated as complete, but somewhat out-of-date since the last update was 1986, while the other types of documentation (e.g., functional requirements; system, database, and program specifications; and program documentation) were rated as nonexistent.

Figure 4-3

More Than Half of the Application Systems are Structured and Modular

Application	Design and Development Methodology
Adopt-A-Horse	Structured and modular. Data flow diagrams, flow charts and N-S diagrams available
Automated Fleet Management	Structured and modular. Code conforms to standards of the Computer Applications Handbook.
Automated Personal Property	Estimated to be 75-80% structured and modular. No diagrams available.
Cadastral Survey Field Note System	Neither structured nor modular.
Checks to Treasury	Structured and modular. Code conforms to standards of the Computer Applications Handbook.
Financial Mngmnt. Year-End Reports	The older programs in this application are neither structured or modular. Newer ones adhere to current application development methodologies and coding standards.
Financial Management Edits	The older programs in this application are neither structured or modular. Newer ones adhere to current application development methodologies and coding standards.
Financial Management Reports	The older programs in this application are neither structured or modular. Newer ones adhere to current application development methodologies and coding standards.
Inventory Data System	Structured and modular.
Lease Management	Design and code were developed using structured coding techniques.
Material Sales	
Operating Budget	COBOL programs conform to BLM's design and coding techniques and are modular.
Program Management	Structured and modular. Code conforms to standards of the Computer

	Applications Handbook.
Reimbursable Billing	Code is fairly well structured and modular.
Summer Hire System	Estimated to be 80% structured and not very modular. No data flow diagrams, though flowcharts are available.
USFS Forest Inventory	Modular and generally structured.
Waterpower System	Neither structured nor modular, though it is reported that the impact of the lack of formal methodolgy is minimal due to the stability of the program's functionality.
Wildfire Reporting	Code is reported as basically in compliance with BLM design and coding standards, but can be more structured.
Wildlife Information System	Structured and modular.

Figure 4-4

The Average Rating for Available Documentation is 3.4

Application	Rating	Comments
Adopt-A-Horse	3	Strong in user and operations documentation. Other types are virtually nonexistent.
Automated Fleet Management	7	Still working on the documentation. Plan on making it very complete.
Automated Personal Property System	4	All types exist, though none are rated well.
Cadastral Survey Field Note System	3	Strong in user documentation.
Checks to Treasury	4	Documentation is still being developed in the user and operational areas. Functional requirements and systems specifications are extensive and up-to-date.
Financial Management Year-End Reports	0	
Financial Management Edits	1	
Financial Management Reports	4	Only programs written within the last five years are well documented.
Inventory Data System	2	Documentation exists for the Ecological Site Inventory System which this application is replacing. Documentation is currently being compiled for IDS.
Lease Management	0	None exists now. Documentation is being written as the new code is being written (converting to a newer version of COBOL) and will be complete.

Material Sales	1	
Operating Budget	4	Very strong in functional specifications and user documentation, but weak in other areas.
Program Management	8	
Reimbursable Billing	7	
Summer Hire System		
USFS Forest Inventory	3	Most of the development specifications have been retained by USFS. BLM has user and operational documentation.
Waterpower System	4	Highly rated functional requirements and system specifications, but weak on program and user documentation.
Wildfire Reporting	5	Some types of documentation were rated well (9), but most were given 4s.
Wildlife Information System	5	Though not complete, an operations manual and user documentation exist for this application. data flow diagrams are available for earlier versions.

4.1.4 Data Standards

As indicated in Figure 4-5, all of the applications written (and newer programs added to older applications) within the last five years (with the exception of the Adopt-A-Horse application) use standard naming conventions for data elements. Some of the older programs (i.e., Cadastral Survey Field Note System and Wildlife Information System) use consistent naming conventions within the application. Two of the applications used by BLM -- the USFS Forest Inventory system and the Waterpower system -- were written outside of BLM; so although consistent naming conventions were used within each application, they do not conform with BLM's standard naming conventions.

Although the Data Element Dictionary (DED) is not used by all the programs in every applications, more than half of the applications use it. Those applications not using the DED apply their own data standards for supportability and consistency. For example, the Checks to Treasury Application does not use the DED, but the application development team maintains their own library of data names and enforces naming conventions.

4.1.5 Test Data

Test data exists for nine of the nineteen applications. As shown in Figure 4-6, five of the applications (Automated Fleet Management, Automated Personal Property System, Inventory Data System, and USFS Forest Inventory) were reported as having test data that tested 100% of the program's logic.

For this set of nineteen applications, 39% of the logic paths are tested by existing test data. FSMSC guidelines recommends that 70% of the logic paths be tested.

4.1.6 Conversion History and Age

Details of the conversion history for these nineteen application systems are contained in Figure 4-7. Most of the applications have no conversion history. Only applications written prior to the installation of

Figure 4-5

Over 73% of the Application Systems Use Standard Naming Conventions

Application	Naming Conventions	Data Element Dictionary
Adopt-A-Horse	n	n
Automated Fleet Management	y	y
Automated Personal Property System	y	y
Cadastral Survey Field Note System	y	some
Checks to Treasury	y	n
Financial Management Year-End Reports	newer pgms	newer pgms
Financial Management Edits	newer pgms	newer pgms
Financial Management Reports	newer pgms	newer pgms
Inventory Data System	y	y
Lease Management	y	y
Material Sales		
Operating Budget	y	n
Program Management	y	some
Reimbursable Billing	y	y
Summer Hire System	n	n
USFS Forest Inventory	n	n
Waterpower System	y	n
Wildfire Reporting	n	y
Wildlife Information System	y	y

Figure 4-6

Test Data Exists for Nine of the Application Systems

Application	Test Data	Percentage of Logic Tested
Adopt-A-Horse	no	
Automated Fleet Management	yes	100
Automated Personal Property System	yes	100
Cadastral Survey Field Note System	no	
Checks to Treasury	yes	95
Financial Management Year-End Reports	no	
Financial Management Edits	no	
Financial Management Reports	no	
Inventory Data System	yes	100
Lease Management	yes	90
Material Sales	no	
Operating Budget	no	
Program Management	no	
Reimbursable Billing	no	
Summer Hire System	yes	85
USFS Forest Inventory	yes	100
Waterpower System	yes	60
Wildfire Reporting	yes	100
Wildlife Informationo System	no	

Figure 4-7

Five Application Systems Have a Conversion History

Application	Approx. Age	Converted
Adopt-A-Horse	1 year	no
Automated Fleet Management	3 months	no - replacing older system
Automated Personal Property System	two years	yes
Cadastral Survey Field Note System	pre-1972	yes - straight code, with no modification
Checks to Treasury	< 1 year	no
Financial Management Year-End Reports	original pgms - 15 years. tailored every year	older pgms
Financial Management Edits	original pgms - 15 years	older pgms
Financial Management Reports	original pgms - 15 years. tailored on a regular basis	older pgms
Inventory Data System	2 years	vegetation data was converted from the Ecological Site Inventory Application.
Lease Management	13 years	yes
Material Sales	some parts - over 10 years others - less than 3	no
Operating Budget	3 years	no
Program Management	2 years	no
Reimbursable Billing	4 years	no

Summer Hire System	6 years	no
USFS Forest Inventory	5-7 years	no
Waterpower System	BLM aquired 5 years ago. Army Corp of Engineers may have dev'd it earlier	no
Wildfire Reporting		no
Wildlife Information System	6 years - original version. 3 years - newest version.	no

the Honeywell computer system in 1978 have been converted. These include the original Financial Management Edits, Reports, and Year-End Reports programs, the Lease Management programs, and the Cadastral Survey Field Note System. These were primarily straight code conversions (no functional improvements, just syntax changes so the programs could be executed in the new environment).

The older COBOL applications (i.e., the original Financial Management Edits, Reports, and Year-End Reports programs, the Lease Management programs) are written in an obsolete version of COBOL. However, the newer programs in the Financial Management Edits, Reports, and Year-End Reports applications are written in the current version of COBOL, COBOL 74. The Lease Management System is currently being totally rewritten in COBOL 74. Also, as mentioned earlier, WO-770 will issue an annual work plan directive to DSC to upgrade its COBOL 68 to COBOL 74 in fiscal year 1988.

4.2 MEETING USER REQUIREMENTS

This section summarizes the comments made by members of the BLM ADP staff concerning how well these nineteen applications are meeting user requirements and the improvements already planned for these applications. These are included, in more detail, in the individual application assessments in Appendix A. In addition, the reader is provided with the cost estimates (generated by the COCOMO model) to completely redevelop these applications. The purpose of including these estimates is to provide BLM with an upper bound for the resources to be spent in modifying an application. More resources should not be expended to modify the application than it would cost to redevelop it.

4.2.1 User Comments

Users do not seem to have major problems with the functionality of these applications. Six of the applications (Inventory Data System, Operating Budget, Reimbursable Billing, Summer Hire System, Wildfire Reporting System, and Wildlife Information System) were reported as having no outstanding change requests at this time. For five of the applications (Adopt-A-Horse, Automated Fleet Management, Automated Personal Property System, USFS Forest Inventory,

and Waterpower System), it was reported that many of the users would like the applications to execute more quickly. In the majority of the cases, it appears to not be isolated to specific application systems but rather a symptomatic problem of the available ADP resources (i.e., processing power, communication capabilities, etc.) which is adversely affecting the desired response time.

Some of the change requests involve only the input/output interfaces. For example, the Cadastral Survey Field Note and Waterpower applications users requested data entry screens, so they can do their own data entry and updates. Also, the Checks to Treasury and Lease Management applications users have requested the ability to print reports on-site.

4.2.2 Planned Improvements

Six of the nineteen applications were reported as having planned improvements. These improvements include:

- Adopt-A-Horse - Converting the application to run on microcomputers so as to eliminate slow response time.

- Automated Fleet Management - Replacing the Motor Vehicle System. This application is still under development. The BLM ADP staff plan to address user comments concerning the Motor Vehicle System as they are developing the Automated Fleet Management System (i.e., additional reports and query options, faster execution).

- Automated Personal Property System - Modifying the application to incorporate automatic input of data from barcode readers so the property numbers can be input more quickly as requested by the users.

- Checks to Treasury - Implementing an indexed-sequential file access method to improve response time.

- Waterpower System - Developing or planning to develop data entry screens so users may do their own input and update. BLM ADP staff intend to acquire portions of this application from the Army Corps of Engineers which is in the process of converting it to operate on a microcomputer.

- Wildlife Reporting System - Converting State and District Office codes from numeric to alphanumeric.

Although there are no specific improvements currently planned for the Cadastral Survey Field Note System, the BLM ADP staff indicated that they would recommend the application be completely rewritten if it is a candidate for modernization.

4.2.3 Redevelopment Costs

The estimated costs for redeveloping these nineteen applications are summarized in Figure 4-8. The effort required to develop these applications was generated by the COCOMO model, and converted into a dollar amount based upon BLM supplied costs per person-month. Appendix B contains the details supporting these estimated costs. As previously stated, these figures should be considered as upper limits when comparing the costs to convert, modify, or redevelop an application system.

Figure 4-8
ESTIMATED COSTS FOR REDEVELOPMENT
OF THE
NINETEEN APPLICATION SYSTEMS

These costs are based on the Development Person-Months estimated by the COCOMO model and a rate of $3,100 per man-month (this figure was provided by BLM).

Development Application	Man-Months	Cost
Adopt-A-Horse	125	$387,500
Automated Fleet Management	292	$905,200
Cadastral Survey Field Notes	41	$127,100
Checks to Treasury	111	$344,100
Financial Management Edits	26	$ 80,600
Financial Management Reporting	66	$204,600
Financial Management Year-End	10	$ 31,000
Inventory Data System	71	$220,100
Lease Management	43	$133,300
Material Sales	91	$282,100
Operating Budget	16	$ 49,600
Program Management	25	$ 77,500
Property Management	91	$282,100
Reimbursable Billing	93	$288,300
Summer Hire System	16	$ 49,600
USFS Forest Inventory	18	$ 55,800
Waterpower System	214	$663,400
Wildfire Reporting	28	$ 86,800
Wildlife Information System	96	$297,600
TOTAL ~~COSTS~~	1,473	$4,566,300

5. FEASIBILITY OF IMPLEMENTING A SIP

This chapter discusses the feasibility of implementing a Software Improvement Program in BLM. The criteria for determining if a SIP is feasible were outlined in Chapter 2. They included:

- Significant investment in existing applications software,
- Software which essentially meets functional needs, and
- Continuing high investment in software maintenance.

In the following three sections, AMS examines whether these criteria are met for BLM's existing applications. In addition, Section 4 outlines the advantages and disadvantages of the three alternatives available to BLM: to leave the applications as they are, redevelop the applications, or implement a SIP to augment BLM's Life Cycle Management methodology in upgrading the existing applications.

5.1 CRITERIA 1: SIGNIFICANT INVESTMENT IN EXISTING APPLICATIONS SOFTWARE

BLM currently has nineteen large Bureau-wide applications that are expected to operate in the new technical environment (i.e., they will not be superceded by another application or discontinued). The total lines of code for these applications is 606,099. They range from approximately 4,350 lines of code to 100,500 lines of code, with a median of 27,072 lines of code. Although using the number of lines of code is a relatively simplistic measure of software investment, these numbers are large enough to indicate a significant investment in existing applications software.

5.2 CRITERIA 2: SOFTWARE WHICH ESSENTIALLY MEETS FUNCTIONAL NEEDS

Based on the comments AMS received from BLM ADP staff in response to our queries about the applications' weaknesses in meeting user needs

(summarized in Chapter 4 and detailed in Appendix A), we concluded that, the existing applications essentially meet the users' functional needs.

Many of the applications were reported as having no outstanding change requests or lacking any functions the users needed. The major requests concentrated on interface improvements, such as on-line access for data entry and query rather than additional software functionality. User requests for faster processing speeds are being answered in some applications by moving them, wholly (e.g., the Adopt-A-Horse application) or partially (Waterpower application) onto microcomputers. In most cases, it is the not the application itself which executes slowly, but the remote computing factor which causes the slower than desired response times.

5.3 CRITERIA 3: CONTINUING HIGH INVESTMENT IN SOFTWARE MAINTENANCE

As described in Chapter 4, the examination of BLM applications with respect to the six areas which affect the maintainability of an application has led us to conclude that BLM is investing a lot of resources in maintaining these applications. These resources are more than comparable more currently developed software would require.

5.3.1 Application Language Usage

As stated in Chapter 4, although the applications primarily conform to ANSI standards for the portions of them that are written in programming languages for which there are ANSI standards (e.g., COBOL and FORTRAN), many of them include additional vendor-specific code. Inclusion of vendor-specific code makes the application dependent on the environment. As a result, BLM requires an ADP staff knowledgeable in each programming language supported by the Bureau increases the resources necessary for application support.

5.3.2 Design and Development Methodology

Most of the Bureau-wide application code was reported to be wholly or partially structured and modular with the exception of the older systems. The

applications written in the last two years conform to the standards of the computer Applications Handbook.

ADP productivity is decreased (and hence support cost increased) for those applications which are not structured or modular. It is much harder to follow a program's logic and, thus, implement and test changes in an unstructured and nonmodular program.

5.3.3 Documentation

Functional requirements; system, database, and program specifications and flowcharts are rated as poor or nonexistent for all applications except the Automated Fleet Management, Checks to Treasury, Inventory Data System, Lease Management (whose documentation is currently being compiled according to the Applications Handbook guidelines), and Waterpower applications. Programmer productivity greatly decreases if each time a program must be modified, it takes days just to determine what the program is supposed to accomplish and what is actually happening.

Most application documentation was not rated as uniformly weak, but rated as weak in some areas and strong in others. Documentation is expected to be thorough for the newer applications undergoing development or rewrite (Automated Fleet Management, Checks to Treasury, Inventory Data, and Lease Management). No applications have complete and up to date documentation at this time.

5.3.4 Data Standards

The majority of the Bureau-wide applications employ uniform naming conventions for the data elements within an application. Less than half use the Data Element Dictionary across all programs in the application (all of the programs in seven applications do, some of the programs in five applications do).

5.3.5 Test Data

Maintenance costs are affected by the testability of software programs. Comprehensive test data files facilitate testing in that they provide the programmer with a means to easily demonstrate the accuracy of program modifications.

The Federal Software Management Support Center (FSMSC) recommends that 70% of a program's logic (at a minimum) be tested by test data. For the nineteen applications considered in Chapter 4, only eight of the applications meet this criteria. Ten of the applications were listed as having no test data at all.

5.3.6 Conversion History and Age

The older BLM programs that were originally written to run on the Burroughs computer system were converted to run on the Honeywell after its acquisition in 1978. The conversions were primarily straight-code conversions. That is, the code was modified to run in the new environment with no enhancements. It is probable that some operations could have been optimized for the new environment, but straight-code conversion does not incorporate rewriting any code.

5.4 ALTERNATIVE ACTIONS FOR BLM'S CONTINUING APPLICATIONS

The applications BLM plans to implement in the new technical environment will be converted to execute on the new computer systems. There are three courses of action BLM may take with respect to transferring these applications to the new environment:

- Essentially leave the application as it is, performing only a straight-code conversion,
- Start from scratch and totally redevelop the application, starting with the development phase of the Life Cycle Management process, or

- Implement a Software Improvement Program (SIP) to upgrade the application and augment BLM's Life Cycle Management process.

The advantages and disadvantages of each of these alternatives are summarized in Figure 5-1. The following four sections provide an analysis of each of these alternatives and our recommendation on SIP feasibility.

We assume the applications are converted to operate in the new technical environment for all alternatives.

5.4.1 Straight-Code Conversion

Software conversion involves transforming application programs and data files -- without changing any program functionality -- so they may be used in a different technical environment. A straight-code conversion precludes the redesign and redevelopment of existing code to enhance the application's functionality or technical performance.

Although this alternative would not use any additional resources (once the conversion is complete), it is not a very good alternative for BLM. During the conversion effort, a lot of information on each application would have been collected in order to define the work packages (see AMS' Software Conversion Study, delivered to BLM June, 1987 for more detail). To not take advantage of all this information, and improve the applications where need be, would be a disadvantage to BLM. The gap between user needs and ADP services will grow if, in the process of transferring applications to the new technical environment, user requests and the applications' technical ability to meet these requests are not taken into consideration.

5.4.2 Redevelop

Under this alternative, BLM's Bureau-wide applications would be redeveloped, starting from the second phase of the Life Cycle Management process, the development stage. The BLM ADP staff would define user needs for each application, design a system to meet those needs, and write, document and test the software programs required.

Figure 5-1

A SIP is Feasible for BLM

Alternatives	Analysis of Alternatives	
	Advantages	Disadvantages
Leave as is	No commitment of resources	Wastes conversion efforts Gap between user needs and ADP services will grow
Start from scratch	Familiar process Avoids trying to understand old code	Many of BLM's systems are deficient technically, not functionally BLM has a significant investment in software
SIP to augment LCM	Helps conserve BLM's software investment First SIP release is similar to the conversion process and could build on that Incremental, so easy to show progress	Unfamiliar process Requires an improved SET

This alternative eliminates the need for programmers to decipher the existing application code. If a program is not structured and has not been well documented, deciphering the code may be very time consuming. Since the LCM process is familiar to BLM ADP staff, this alternative avoids the expense of training the staff in a new methodology. The new code should be easier to maintain. It would be state-of-the-art and newly documented (and documented thoroughly, if all documentation required by BLM ADS development guidelines are provided).

One disadvantage of this alternative is that BLM has a significant investment in software applications, many of which were reported to be technically obsolete but functionally sufficient. Except for a few requests for additional functionality (e.g., adding a program to the Checks to Treasury application for manual payment schedules or providing users of the Cadastral Field Notes Survey and Waterpower applications the ability to enter data on-line), BLM's existing application systems are meeting the major functional needs of the users. In these cases, defining the functional requirements from scratch is not necessary and a waste of resources.

5.4.3 SIP to Augment LCM

Under this alternative a Software Improvement Program, as defined in Chapter 2, would be implemented to augment BLM's Life Cycle Management process. Software Improvement is defined by the Federal Conversion Support Center (FCSC) as "preventive maintenance". The software is "cleaned-up" to increase its reliability, efficiency, portability, and maintainability in order to keep it in working order and capable of fulfilling present and future ADP requirements. Software is not only changed in response to user requests for adjustments or enhancements, but in anticipation of changing needs.

This alternative helps conserve BLM's investment in its application software. The current software performs the functions required by BLM to perform its mission and thus has an intrinsic value to the Bureau because it is already working and tested. A SIP provides the means by which to determine, on an application by application basis, the actions required to modify and/or implement the converted applications (e.g., salvage the existing

software by leaving it as it is or by improving it through refinement or enhancement, replace the software with an external software package, or redesign/newly develop the application software).

The SIP would be an extension of BLM's conversion effort. As we mentioned in Chapter 2, conversion is the first release of a SIP. Conversion, however, under a SIP is not straight-code conversion. In the process of converting code under a SIP also includes activities for standardizing code, data names, and format; upgrading programming languages and operating systems; translating low-level languages to high-level languages; and removing environmental dependencies (e.g., removing machine-dependent code and replacing it with more portable code). Thus the SIP would build on work BLM has already planned.

In addition to the advantages outlined in the past two paragraphs, the Software Improvement approach also provides for orderly, incremental improvement of an organization's application software. The process is iterative in nature. It's incremental approach allows for testing small, manageable "pieces" which provide constant achievement, growth and progress feedback to both the application developers and users. A usable version of the new or improved system would be available at each stage of development.

The primary disadvantage of this alternative is that the BLM ADP staff are unfamiliar with the Software Improvement process. The ADP staff must therefore be trained in this methodology. In addition, a SIP requires an established modern SET (the Software Engineering Technology described in Chapter 2). Development and implementation of an improved SET, in itself, is resource intensive. BLM, however, would benefit directly from an improved and synchronized set of software standards and guidelines, procedures, tools, quality assurance, and training policies provided by a SET.

5.4.4 Recommendation of SIP Feasibility

We have examined the Bureau-wide, Honeywell-based applications. Although some will be superceded by BLM or DOI systems, the remaining application systems still represent a significant software investment. In light of the improved delivery of ADP modernization benefits a SIP would

provide to users and OMB maintenance cost objectives, it is AMS' recommendation that the best alternative for BLM's applications is to implement a Software Improvement Program. A SIP should make better use of the results of the efforts required for conversion to the new environment and preserve the value of the Bureau's past software investments. At the same time BLM increases the quality of its application software to create an improved foundation upon which to maintain existing applications and develop new ones.

6. ENSUING STEPS

This document has recommended BLM initiate a SIP in the implementation of it's modernization program. BLM, however, must take some additional steps before initiating an SIP. AMS will also use the results of the software assessment as input into a remaining task of the ADEMP project. This chapter describes the steps remaining in the ADEMP project for AMS and identifies and recommends tasks BLM should initiate for implementing an SIP.

6.1 SUBSEQUENT TASKS FOR THE ADEMP PROJECT

AMS has three remaining tasks in the ADEMP project:

- Task 8, Economic Analysis of Feasible Alternatives
- Task 9, Implementation Strategy
- Task 10, Technical Specifications

Of these, only the Technical Specifications task relies on the software assessment data.

During these tasks, AMS will cost representative technical architectures and identify the one which represents the best economic alternative, present an implementation strategy for installing this architecture, and then develop the technical specifications for its procurement.

6.1.1 Economic Analysis of Feasible Alternatives

In Task 8, the Economic Analysis of Feasible Alternatives, we will present the costs for the remaining alternative architectures and select a model architecture based on these costs. The per unit costs we use will be based on modeling the projected processing requirements utilizing industry estimates, National Bureau of Standards projections, AMS experience, and BLM guidance. Our workload estimates will be based on the results of prior ADEMP

tasks and the results other BLM-conducted studies, such as the ALMRS Feasibility Study, the ARRS Requirements Study, the LIS RFC, and the OA Requirements Study. Overall costs will be derived by sizing the configuration required to handle the projected workload under a specific alternative and applying the per unit costs to that configuration. We will select a model architecture based on costs and other relevant but less tangible factors such as ease of administration and reliability.

We will use the model architecture as a base for estimating the cost to BLM but not to dictate the specific technical solution for inclusion in the Technical Specifications (Section C of an RFP). The Federal oversight agencies (i.e. GSA, OMB, GAO) generally prefer a functional specifications approach over the specification of a specific technical solution. It helps mitigate claims of vendor specific bias and usually encourages competition. It also encourages vendors to propose innovative solutions which make the best use of their products.

Of course, BLM cannot estimate its costs based on functional definitions. It must have a model architecture which represents the best potential solutions vendors may propose. This architecture must consider the issues facing BLM and include the components BLM may require in the size and quantity necessary.

This model architecture may not be the "optimum" solution. Vendors can propose solutions which are different than the model architecture but more effective or technically robust because they play to the strengths and interrelationships of vendor-specific products. However, BLM could not specify these solutions in an RFP because they are vendor-specific and so may restrict competition. Hence, we will identify and cost the best generic architecture which provides a good approximation of what vendors are likely to bid.

6.1.2 Implementation Strategy

During Task 9, the Implementation Strategy, we will identify and analyze the issues involved in selecting and implementing the ADEMP. These

issues will include identifying the potential procurement strategies, selecting the best alternative strategy, suggest an evaluation methodology to select the most competitive proposal(s), discuss the implementation schedule, recommend potential steps/tasks in the plan, and identify and recommend an organizational structure to implement the plan.

An organization's implementation objectives determine which issues take precedence in its implementation strategy. For example, the implementation sequence and timeframe effects the number of implementation teams, their training, and the level of synchronization required. On the other hand, reliability requirements may effect the length of the parallel processing period and the rigor of the testing process.

In our Implementation Strategy report, we will describe the issues of most importance to BLM and the strategy we believe is most appropriate for ADEMP. We will also address how this strategy relates to other efforts within BLM in terms of timing and other impacts.

6.1.3 Technical Specifications

In Task 10, the Technical Specifications, we will provide a document which is essentially a draft of the ADEMP study's components of BLM's technical architecture for use as Section C of the ADEMP RFP(s). These specifications will be based on the components identified in the model technical architecture and on BLM direction concerning both the number of procurements and their scope.

6.2 TASKS FOR BLM TO COMPLETE PRIOR TO IMPLEMENTING AN SIP

Before implementing a Software Improvement Program for the Bureau's applications, we recommend BLM conduct a SIP pilot study to test the SIP concept and develop the SET necessary to support a full SIP. An SIP pilot applies the SIP methodology to a reduced set of applications, usually defined by a specific functional area. To adequately define and select this set of applications requires the same initial planning steps as a full SIP.

Basic to such a pilot is an understanding of the SIP planning process and an understanding of the criteria applicable to selecting applications for improvement. The following sections contain a brief description of the SIP planning process and identifies four application improvement criteria. For a detailed description of the SIP planning process the reader is referred to two GSA documents published by the Office of Software Development, Federal Conversion Support Center: Guidelines for Planning and Implementing a Software Improvement Program (SIP) and The Software Improvement Process - Its Phases and Tasks.

6.2.1 Overview of the SIP Planning Process

A comprehensive plan is a prerequisite of a successful SIP. Thorough planning reduces risk and establishes a framework for SIP direction and management controls.

Planning for the Software Improvement process is performed in a hierarchical, sequential fashion. Higher-level plans (i.e., macroplans) influence or control lower-level plans (i.e., microplans or software improvement release specifications).

The plan deals with the elements of a SIP: increments, releases, and phases. Increments are the limited improvements planned for each application during the current iteration of the plan. Releases are groups of increments scheduled for implementation at the same time and usually focused on the same or similar objectives. For example, a conversion release would be targeted to making the applications included as transportable as possible. Phases are the steps of the SIP process which determine which increments and releases are appropriate at a given time.

The SIP planning process is both incremental and iterative. Throughout the process, the SIP team periodically refine the macro- and microplans until the plans are formalized as specific, detailed work plans and schedules. After each improvement increment and release, the team reviews the results and methodologies used to provide feedback to the original macro- and microplans. The main objective of the assessment of each completed improvement effort is

to provide strategic direction to subsequent improvement increments and releases. That is, subsequent increments and releases may be updated or revised to reflect significant accomplishments or changes in direction as determined from assessing the completed improvement increment or release.

Sample macro- and microplan outlines are provided in Figures 6-1 and 6-2. The macroplans themselves impose boundaries on subsequent microplans. They should describe, in as much detail as possible, the increments of the SIP (and the increment's releases, if possible at this time). The macroplans should also describe the applicable software improvement process phases and tasks required to implement the SIP. They should define the overall goals and objectives, assumptions and constraints, and strategies for the SIP. The purpose of the microplans is to provide detailed instructions on how BLM intends to accomplish the improvement(s) for each increment and its releases. A separate microplan is developed for each increment. As evident from the sample outline in Figure 6-2, the microplan must document the improvement strategy for the increment and the task and responsibility descriptions and assignments.

6.2.2 Criteria For Application Improvement

As part of the SIP planning process, BLM must determine which software applications are candidates for improvement. Four criteria are key in determining whether or not an application should be improved. These are:

- Application size,
- Volatility,
- High Leverage, and
- Mission/criticality.

In this analysis, we have discussed BLM's applications based on the first criteria, application size. It is the only criteria which can be assessed objectively. The other three all require applying subjective judgements based on in-depth knowledge of the specific application itself or of BLM priorities.

Figure 6-1
Sample SIP Macro Plan Outline

1. INTRODUCTION
    - 1.1 Description and Scope of Work for SIP
    - 1.2 Content and Purpose of Macroplan
    - 1.3 Background and History Information
    - 1.4 Need for Software Improvement
    - 1.5 Objectives of SIP
    - 1.6 Major Assumptions and Constraints
    - 1.7 Points of Contact
2. DESCRIPTION OF ENVIRONMENT
    - 2.1 Description of Current Environment
        - 2.1.1 Hardware
        - 2.1.2 Software
        - 2.1.3 Operating Environment
        - 2.1.4 Work Methodologies
        - 2.1.5 Users
    - 2.2 Description of Future Environment (Concept of Operations)
        - 2.2.1 Hardware
        - 2.2.2 Software
        - 2.2.3 Operating Environment
        - 2.2.4 Work Methodologies
        - 2.2.5 Users
3. PROJECT INITIATION
    - 3.1 Software Improvement Strategy
        - 3.1.1 Approach
        - 3.1.2 Methodologies/Techniques
        - 3.1.3 Software Engineering Technology (SET) Baseline
    - 3.2 SIP Organization
        - 3.2.1 Structure
        - 3.2.2 Personnel Responsibilities and Authorities
    - 3.3 Planning Considerations

Figure 6-1, Cont'd
Sample SIP Macro Plan Outline

4. GUIDELINES FOR SIP PROCESS
   - 4.1 Description of Software Improvement Process (Phases and Tasks)
   - 4.2 Resources
     - 4.2.1 Estimates
     - 4.2.2 Schedules
   - 4.3 Microplan Development
     - 4.3.1 Content and Format
     - 4.3.2 Level of Detail

5. RECOMMENDATIONS AND CONCLUSIONS
   - 5.1 Description of Pilot SIP Project
   - 5.2 Generic Software Improvement Recommendations
   - 5.3 Recommended SIP Implementation Approach

6. SIGNIFICANT OCCURRENCES/ACCOMPLISHMENTS
   (Add sections periodically)

APPENDICES

- MICROPLANS for each increment
- Significant or important documents
- Detailed project schedules and cost analyses

Figure 6-2
Sample SIP Micro Plan Outline

1. INTRODUCTION
    - 1.1 Description and Scope of Work for Increment
    - 1.2 Content and Purpose of Microplan
    - 1.3 Objectives of SIP for Increment
    - 1.4 Major Assumptions and Constraints
    - 1.5 Points of Contact

2. IMPROVEMENT STRATEGY FOR INCREMENT
    - 2.1 Description of Increment's Releases
    - 2.2 SIP Organizational Assignments
    - 2.3 Planning Considerations
    - 2.4 Modifications to Baseline SET

3. SOFTWARE IMPROVEMENT TASK ASSIGNMENTS FOR INCREMENT
    - 3.1 Task 1 - Inventory and Analysis
        - 3.1.1 Task Description
        - 3.1.2 Resource Estimates and Allocation
        - 3.1.3 Schedules

        (Repeat same components 3.2 through 3.11)
    - 3.2 Task 2 - SIP Plan Development
    - 3.3 Task 3 - Establishment of Engineering Elements
    - 3.4 Task 4 - Work Package Preparation
    - 3.5 Task 5 - Test Data Set Preparation
    - 3.6 Task 6 - Improvement Release Specifications Preparation
    - 3.7 Task 7 - Software Improvement
    - 3.8 Task 8 - Testing
    - 3.9 Task 9 - Documentation
    - 3.10 Task 10 - Acceptance Testing
    - 3.11 Task 11 - System Transition
    - 3.12 Summary
        - 3.12.1 Summary of SIP Task Estimates
        - 3.12.2 Summary of SIP Task Schedules

Figure 6-2, Cont'd
Sample SIP Micro Plan Outline

4. SIGNIFICANT OCCURRENCES/ACCOMPLISHMENTS
(Add sections periodically)

APPENDICES

- SOFTWARE IMPROVEMENT RELEASE SPECIFICATIONS for each release
- Detailed schedules and cost analyses for each release
- Significant or important documents

6.2.2.1 Criteria 1: Application Size

Large applications (in terms of lines of code) generally consume more resources for maintenance and support than do smaller ones. In addition, the size of an application is a telling, though somewhat simplistic, measure of the amount of resources invested in the application's development.

A SIP attempts to reduce maintenance costs while preserving existing software investment and avoiding redevelopment costs. Given this, a logical first place to identify SIP candidates is where maintenance costs are likely to be high and ADP staff resource investments are high. Large applications tend to be higher in both of these areas. Size alone, however, is not necessarily an indication that an application requires improvement.

6.2.2.2 Criteria 2: Volatility

Highly volatile applications may also increase maintenance costs. Volatility refers to the amount of maintenance or change required by an application. Applications must be designed to handle high volatility if that is the environment in which they will operate. Note that volatility is independent of application size.

Ease-of-Maintenance was a low priority in earlier application development approaches. Other resources such as CPU memory required and execution time were more limiting than maintenance resources. Therefore they received more attention. Maintainability issues were largely ignored. Current methodologies place more emphasis on techniques which reduce maintenance as this cost has been identified as the largest cost factor for an application over its life-cycle.

In a highly volatile environment, the maintainability of an application comes into play more often. One may be able to afford a difficult-to-maintain application if it changes infrequently. However, in a volatile environment, maintainability quickly becomes a primary objective.

Therefore, if an application is modified/enhanced frequently and the processes to accomplish this are somewhat difficult and costly, it should be considered as a strong candidate for improvement. This analysis will require knowledge of the application's change history and detailed familiarity with the application code.

6.2.2.3 Criteria 3: High Leverage

By "High Leverage" we mean an application where significant improvement in performance could be realized with only a small amount of work. For example, an application with an inefficient, custom-written sort routine may be greatly improved by simply removing the sort code and using a more efficient external sort package. This criteria is independent of application size and volatility.

High leverage applications can be identified by their use of unusually large amounts of one or more kinds of ADP resources (e.g. print pages, CPU seconds, or disk storage). To be high leverage, there has to be the potential for large savings of some kind. This resource consumption, however, must be high relative to the job the application is performing. For example, one would expect a scientific application to use more internal memory than a business application but a comparable amount of memory to that of another similar scientific application.

Applications meeting the high leverage criteria are very valuable because they can quickly establish the value of the improvement program to the organization. Users see immediate and often dramatic results without significant time or effort investment by the ADP staff.

Identifying a high leverage application can be difficult. It requires familiarity with both the application's resource usage and the relative resource usage of comparable applications in the same environment.

6.2.2.4 Criteria 4: Mission Critical

By "Mission Critical", we mean applications which are pivotal to the mission of part or all of the organization. If these applications fail or are disrupted, the ability of the organization to perform its mission is in jeopardy.

Some organizations are so dependent on computers that they could not perform their mission without certain key applications. For example, the Federal Housing Authority (FHA) could not possible process all the mortgage applications they receive by hand. For these applications, the organization wants to ensure that changes can be incorporated with the minimum possible risk of disruption. This includes the use of modern application design and coding techniques where maintainability is a high priority.

Determining which applications are most critical can also be difficult. Large organizations are usually comprised of smaller units which each have their own missions and mission critical applications. Upper management must provide the overall perspective necessary to determine the relative criticality of the various applications and enforce decisions based on that assessment.

An application that is critical to the mission of BLM or components of BLM would be a key candidate for improvement. It would be up to BLM management to determine what would make one application "more critical" to BLM's mission than another.

6.3 SUMMARY

This chapter has identified the ensuing steps we recommend for implementing a SIP. If implemented, the resulting costs for maintenance of BLM's current Honeywell-based applications should be reduced, enhancements to these systems completed in shorter periods of time, and system performance improved. We strongly recommend BLM validate these benefits through a pilot study as discussed in Section 6.2.

APPENDIX A

GUIDELINES FOR PLANNING AND IMPLEMENTING A SOFTWARE IMPROVEMENT PROGAM (SIP)

GSA FCSC REPORT OSD/FCSC-83/004

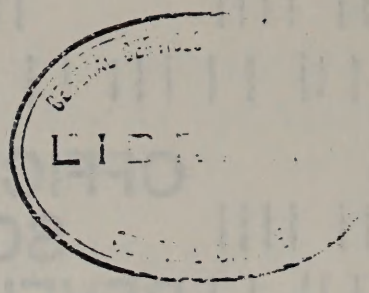
OFFICE OF SOFTWARE DEVELOPMENT

Federal Conversion Support Center

GUIDELINES FOR PLANNING AND IMPLEMENTING A SOFTWARE IMPROVEMENT PROGRAM (SIP)

Report OSD/FCSC-83/004

GSA



Prepared by:

U.S. FEDERAL CONVERSION SUPPORT CENTER
Office of Software Development
Two Skyline Place, Suite 1100
5203 Leesburg Pike
Falls Church, VA 22041

May 1983

GUIDELINES FOR PLANNING AND IMPLEMENTING A SOFTWARE IMPROVEMENT PROGRAM (SIP)

Report OSD/FCSC-83/004

FOREWORD

Over the past several decades, substantial changes have occurred in the automatic data processing (ADP) industry. There have been dramatic increases in hardware productivity [1], with a significant decrease in the "footprint" of the hardware configuration due to its reduced size, component modularity, and lowered air-conditioning and electrical consumption. Simultaneously, total ADP costs have continued to rise with the largest costs shifting from hardware to software. This shift in costs is primarily due to substantial automatic data processing equipment (ADPE) price reductions, coupled with increased personnel costs for software development and maintenance activities.

During this same period of high-powered, low-cost, rapidly-advancing ADPE, many Federal ADP organizations are facing a "software crisis." Software activities are still labor intensive, with little increase in productivity being realized in software production and maintenance. Resource utilization has shifted from software development activities to maintenance, with over half of all software personnel involved in correcting software errors, modifying software to change its functions or extend its life, and simply keeping yesterday's software operational [1].

ADP organizations are plagued with high maintenance costs, long delays in responding to user's changing needs, and continued development and operation of antiquated and underpowered computer software. Productivity increases for these ADP organizations are severely limited due to the proliferation of archaic software analysis, design, coding, and testing features and techniques; low-level and nonstandard languages; machine or environment dependencies; and custom-written utilities.

A reversal of this situation requires an organization's commitment and the establishment of a "software improvement program" (SIP). A SIP is an incremental and evolutionary approach to the modernization of software to maximize its value, quality, efficiency, and effectiveness. A SIP preserves the value of past software investments, and, at the same time, increases the quality of the software to create an improved foundation upon which to maintain older systems and build new ones.

A SIP can be thought of as a "preventive maintenance" program. Like ADPE preventive maintenance, software must also be periodically and systematically cleaned-up, fine-tuned, optimized, and enhanced to keep it in working order and capable of fulfilling its current and future requirements. Thus, the goals of a SIP are many -- to improve software maintenance and control, reduce delays in responding to user's needs, improve software quality, enable more efficient and effective programmer productivity, decrease high software maintenance costs, institutionalize processes, and put the organization in a position where it can take advantage of new and emerging technologies.

EXECUTIVE SUMMARY

This document presents an overview for establishing, planning, and implementing a SIP, to guide ADP managers responsible for performing those tasks. The objective of this document is to focus attention on the key role of timely and thorough planning for a SIP to ensure successful improvements. This document presents SIP planning and implementation concepts, strategies, and considerations, and leads into the more detailed planning required for the SI process, which is described in another document, "The Software Improvement Process --Its Phases and Tasks."

The guidelines presented in this document serve as a starting point in the establishment of a SIP; emphasize the innovative, top-down, incremental approach to planning for and implementing a SIP; and stress the importance of developing a SIP plan. Though general in nature, this document emphasizes "what needs to be done" rather than "how to accomplish" a SIP. As such, a SIP plan, when formulated, should be flexible enough to allow for incorporation of new information and technical and managerial innovations, as they present themselves.

It should be stressed that these SIP planning guidelines work in tandem with the guidelines presented in the recent Federal Software Testing Center (FSTC) document, "Establishing a Software Engineering Technology" [2]. The development and institutionalization of a modern Software Engineering Technology (SET), consisting of a synchronized set of software standards and guidelines, procedures, tools, quality assurance (QA), and training, implemented through and coupled with a dynamic, ongoing SIP are paramount to achieve successful, worthwhile, and long-lasting software improvements. The use of these SIP planning guidelines, in concert with the SET guidelines, is strongly encouraged to promote a more uniform, thorough, and better-planned and -managed SIP.

TABLE OF CONTENTS

LIST OF EXHIBITS

1. INTRODUCTION

1.1 Need for Software Improvement

Most existing Government software is well over a decade old, with some as much as 20 to 25 years old. Much of the software was originally written on second-generation hardware and operating systems, written in machine-dependent and nonstandard languages, and have undergone several hardware, operating system, and language conversions. The majority of this software was written with little or no utilization of software design, programming, or testing standards, guidelines, or procedures; required substantial operator intervention; utilized sequentially accessed card and tape input and output files; and had minimal, inadequate, or in some cases, a total lack of documentation.

Embedded in this aging software are "home-grown" system utility and operating system features such as sorts, merges, record buffers, copies, and manual restarts. These features were included, of necessity, because most of the features of modern software package utilities and operating systems, which we now take for granted, were not available as packages or in operating systems of that day. Many of these home-grown utilities and operating systems are no longer supported by the developing organization or the vendor, nor are there enough adequate programmers available to maintain this software.

In the past, bigger or more powerful ADPE configurations, or emulation or simulation, have been the "quick fix" for these software problems. But increasingly, this hardware fix for software "aches and pains" has been found to be a fleeting panacea, or, at best, a temporary solution, and today's modern systems cannot, and do not, support emulation or simulation of the older programming features and practices.

In addition to these problems of aging software, ADP organizations are plagued with high maintenance costs, long delays in responding to user's changing needs, and continued development and operation of antiquated and underpowered computer software. Productivity increases for ADP organizations with these problems are severely limited, if not impossible to attain, due to the proliferation of archaic software analysis, design, coding, and testing features and techniques; low-level and nonstandard languages; machine or environment dependencies; and custom-written utilities.

This antiquated, aging, outmoded, and relatively "obsolete" softw[illegible] is in need of modernization. While this software cannot be termed totally obsolete, because it is still operational, it can be thought of as being in an advanced state of "software senility." Software senility is a degenerative condition, which, if not corrected, will eventually render the software totally useless.

In view of the many and complex, aforementioned software problems, and the emerging trend that the "software crisis" will continue to grow and worsen, a quick fix or single solution to the problems is not feasible, and a direct conversion from the problem environment to a modern ADPE system and environment is virtually impossible. To solve these problems and combat the software crisis, a program must be instituted to preserve the value of past software investments, as much as possible, and provide an incremental and evolutionary approach to modernizing the existing software to maximize its value, quality, effectiveness, and efficiency.

Such a "software improvement program" (SIP) is described herein as a "treatment" for the ills of software senility, and offers a cure for many of the software problems from which today's Government ADP organizations are suffering. Institutionalization of a sound "software engineering technology" (SET), coupled with a dynamic, ongoing SIP, can attack the causative factors of the software crisis and provide the Government with viable, modernized, effective, efficient, and high quality ADP systems capable of capitalizing on today's modern ADP technology, as well as future technological advances in the field.

1.2 Purpose and Scope of this Document

The objective of this document is to focus attention on the key role of timely and thorough planning for a SIP to ensure successful improvements. This document presents preliminary SIP planning and implementation concepts, strategies, and considerations, and leads into the more detailed planning required for the software improvement process, which is described in another document, "The Software Improvement Process -- Its Phases and Tasks."

This document serves as a framework or starting point in establishing, planning, and implementing a unique SIP. It describes the SIP planning process and the tasks required therein, emphasizing "what needs to be done" rather than "how to accomplish" a SIP.

The guidelines presented in this document emphasize the innovative, top-down, incremental approach to planning for and implementing a SIP and stress the importance of developing a SIP plan. The guidelines describe the software improvement concept, approach, and process, as well as SIP planning and implementation factors, in enough detail for the reader to perform the required planning activities, including making informed and appropriate trade-off decisions for their particular SIP.

1.3 Content of this Document

Section 1 of this document is an introduction, which includes a description of the purpose, scope, and content of this document. Section 2 provides an overview of the SIP, including a description of its concept, approach, process, goals, and benefits. Section 3 addresses the key role of SIP planning and describes the suggested content, scope, and format of a SIP plan. It also discusses planning considerations and major factors that affect SIP planning. Section 4 summarizes the SIP planning process and offers a recommended approach to implementing a SIP.

In addition to the main body of this document, there are several additional sections of importance. A list of specific references, from which pertinent information on software improvement, planning, technological advances, etc., was gathered, is included. Also included is a glossary of terms, including acronyms and their full meaning, and pertinent SIP-related terms and their definitions.

2. OVERVIEW OF THE SOFTWARE IMPROVEMENT PROGRAM (SIP)

The establishment of a SIP for improving existing systems or constructing new systems, in a "building block" fashion, from existing software is a commitment to-

- combat the software crisis;
- resolve existing software problems and/or improve the status quo;
- include software in the overall long-range ADP plans along with ADPE and user-service levels;
- extend the software's life;
- maximize the value, quality, efficiency, and effectiveness of the existing software; and
- change software development and maintenance from a reactive state, where software is only changed in response to ADPE configuration changes or user requests for correction or enhancement, to a proactive state, where software needs are anticipated and planned for well in advance of ADPE changes or user requests, with alternatives for future courses of action offered to ADP personnel and users based upon the software status.

2.1 SIP Description

A SIP is an incremental and evolutionary approach to the modernization of existing software to maximize its value, quality, effectiveness, and efficiency. Modernization includes those activities necessary to upgrade the software and its engineering techniques to current or state-of-the-art levels. A SIP preserves the value of past software investments, and, at the same time, increases the reliability, efficiency, portability, and maintainability of the software to create an improved foundation upon which to maintain older systems and build new systems.

Hence, a SIP can be thought of as a "preventive maintenance" program for software. Like ADPE preventive maintenance, software must also be periodically and systematically cleaned-up, fine-tuned, optimized, and enhanced to keep it in working order and capable of fulfilling current and future requirements.

2.1.1 Concept

The concept of software improvement is not new, rather it is an outgrowth of normal day-to-day software maintenance projects and an extension of conversion projects. Some types of software improvement are presently performed concurrently with such everyday tasks

as software modification or maintenance. Examples of these everyday improvements include realigning code to enhance readability, and implementing naming conventions to facilitate understandability. However, these improvements are traditionally performed on a random, piecemeal basis, without structure or overall software or organizational considerations.

Such improvement decisions are usually made by the individual programmer without the benefit of managerial input. This bottom-up, piecemeal approach to software improvement is unstructured, and usually results in an unsuccessful attempt to cohesively improve the current software and acquire and use modern tools and techniques.

This traditional, single-purpose software improvement approach has also been applied to conversion projects. Typically, conversions are performed due to hardware changes, operating system changes, language upgrades or dialect modifications, or combinations of the above. The amount of improvement made to converted software is usually dependent upon the source and target environment compatibility. If a noncompatible conversion is undertaken, there is greater opportunity for improvements to be implemented than if a compatible conversion is undertaken because extensive improvements are necessary to fit the software into the constraints of the target environment (e.g., standardizing the software code, removing environment and architecture dependencies, and removing archaic coding features). However, since noncompatible conversions can be extremely complex, the improvements made are typically kept to a minimum with most of the effort concentrated on making the converted software operational in the new environment.

In contrast to the traditional, single-purpose software improvement approach, the modern software improvement approach, as described hereinafter, actually serves multiple purposes. Under the modern software improvement approach, improvements are not performed to meet only a single need or objective, but rather to accomplish several objectives and reconcile multiple problem areas. Also, the decisions as to when software improvement is needed and what types of improvements are needed are not left to the individual programmer or analyst, and the improvements are not performed in a casual manner. Rather, these decisions and the improvement performance are institutionalized as a formal process to which all programmers and analysts must adhere.

2.1.2 Approach to Building or Improving Systems

While the software improvement concept may not be new, the software improvement approach to building or improving systems, with its progression through discrete phases of software development (e.g., Feasibility Analysis, Requirements Definition and Analysis Stage, Design Stage, Programming Stage, Validation Stage, Operations Stage, and Review Stage [2]), is innovative and more sophisticated than the

conventional and more simplistic software life-cycle approach. The software improvement approach to building and improving systems [1] is built on the key assumptions that-

- most major ADP organizations have a decade or more of investment in software;
- most Federal organizations are almost entirely dependent on their software to meet their mission;
- keeping software operational is difficult enough without deviating from that baseline of software to add enhancements or change functions through major redesign or new development, which is thought to be an uncertain and risky business; and
- there is a need to support new applications to keep ADP costs low and service levels high.

The software improvement approach to improving existing systems, or building new systems from existing software, is different from the conventional system development approach in that it recognizes-

- the existence and characteristics of the current systems that support day-to-day operations;
- the existence of other operational systems that may be integrated to replace functions in an existing system;
- the inherent problems in engineering new code in any quantity;
- that existing systems are frequently the only specification of existing processes;
- the need to preserve the testing integrity of the current system while moving to a new or improved system;
- that many faults or deficiencies in an existing system can be accurately and cost-effectively corrected by improving it;
- the need for an orderly, incremental approach to the building or improving of a system that allows for adequate testing, manageable pieces, constant achievement and growth, and progress feedback;
- the need for a useable version of the new or improved system at each stage of development or improvement, allowing for rapid capitalization upon the new system and its components; and

- the virtual impossibility of completely re-engineering or redeveloping very large systems within a reasonable time frame [3].

The universe of software, from which a desired application can be built or improved on, can be conceived as a triangle, as illustrated in Exhibit 1.

At the apex of the triangle is all of the "software that currently exists" in production today. This software performs the functions that the organization needs to conduct its day-to-day business. Because this software is already working and tested, it has an intrinsic value to the organization and represents the vested interest an organization has in its own software applications. Existing software is usually salvaged, transferred, and incorporated into a new or improved system by purging any undesirable or unnecessary software, leaving some of the software as it is, and improving the remaining software through conversion, refinement, and enhancement activities. While it is typically the easiest and most accurate to test and the least costly to produce, this software is often the most expensive code to maintain because it is usually undocumented and built in a "patchwork" fashion.

At the bottom left-hand corner of the triangle is "other operational software that exists" in other organizations. This external software represents the software packages available from industry, the Federal Sofware Exchange Center (FSEC), the Department of Energy's Argonne Library, as well as from such informal sources as user's groups or friends. While this software may suffer from some deficiencies, it may be modified to fit the organization's needs. Also, many software packages are highly maintainable, well documented, and quite portable, and may not require extensive, if any, modification. External software is usually incorporated into a system by replacing existing code with an existing package. This software is typically somewhere between existing and new software in cost, accuracy, and maintainability, depending on the package's functions and its level of sophistication.

Finally, at the bottom right-hand corner is "new software," which does not yet exist and must therefore be engineered. While this code is typically the easiest to maintain because it is state-of-the-art and newly documented, in terms of accuracy it is normally the most difficult to engineer and the most risky to undertake because there is no existing baseline from which to test or measure. It is also the most costly to produce because it must be engineered from "scratch." This software should be incorporated into a system as a last resort, if transfer of the existing software or replacement with a package is not feasible. Nevertheless, any new software should be engineered using modern programming practices to ensure software that is well documented; fits the application better; is easy to support, read, understand, modify, and enhance; and is less expensive and time consuming to maintain.

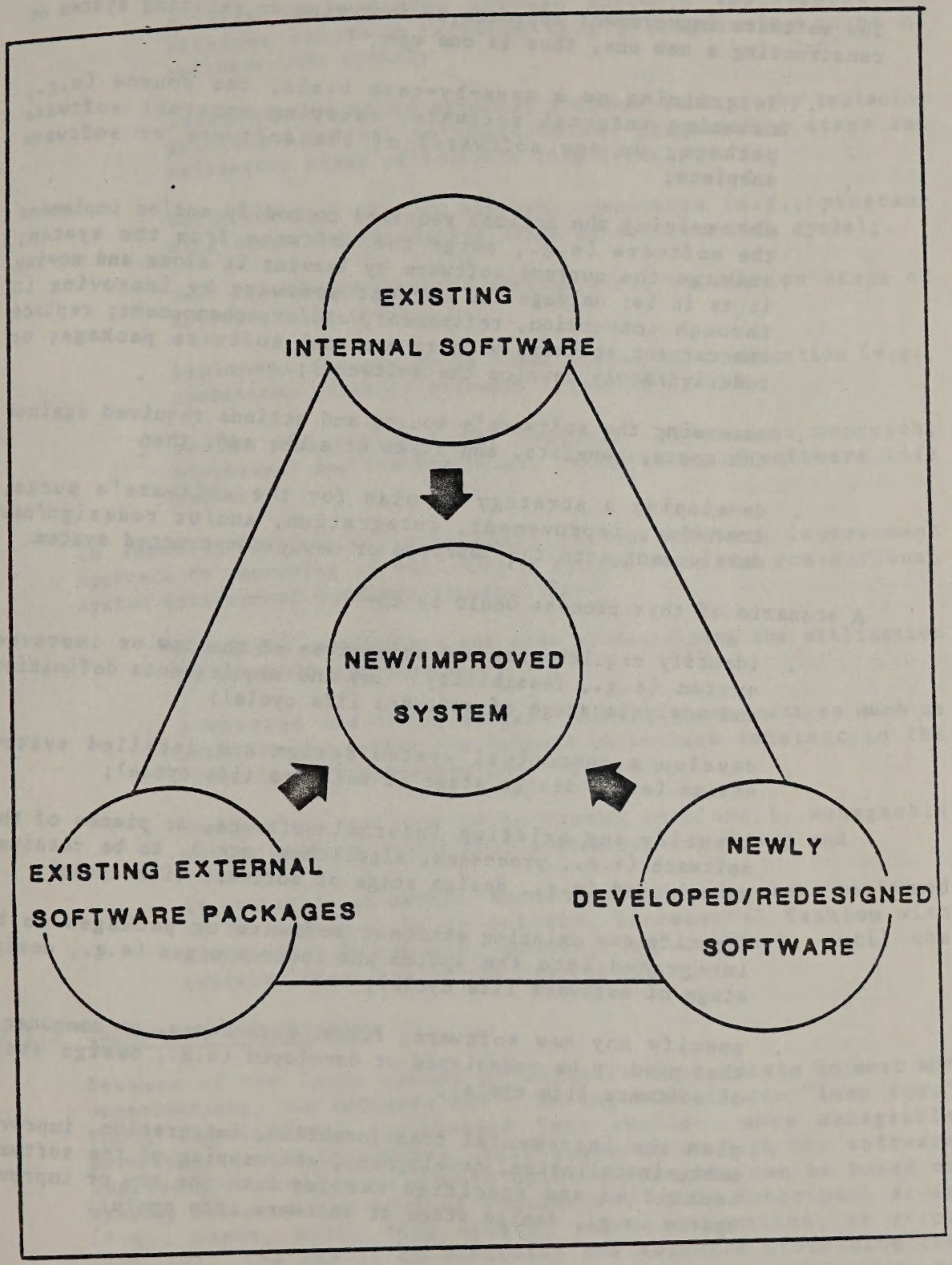


EXHIBIT 1: Software Improvement Approach

The software improvement approach to improving an existing system or constructing a new one, thus is one of-

- determining on a case-by-case basis, the source (e.g., existing internal software, existing external software package, or new software) of the software or software subpiece;

- determining the actions required to modify and/or implement the software (e.g., purge the software from the system; salvage the current software by leaving it alone and moving it as it is; salvage the current software by improving it through conversion, refinement, and/or enhancement; replace the current software with an external software package; or redesign/newly develop the software);

- assessing the software's source and actions required against the costs, benefits, and risks of each; and, then

- developing a strategy or plan for the software's purge, transfer, improvement, integration, and/or redesign/new development into the improved or newly constructed system.

A scenario of this process would be to-

- identify requirements and objectives of the new or improved system (e.g., feasibility study and requirements definition and analysis stage of software life cycle);

- develop a conceptual system design and detailed system design (e.g., design stage of software life cycle);

- identify any existing internal software, or pieces of the software (i.e., processes, algorithms, etc.), to be retained or salvaged (e.g., design stage of software life cycle);

- identify any existing external software or packages to be integrated into the system and their sources (e.g., design stage of software life cycle);

- specify any new software, files, interfaces, or components that need to be redesigned or developed (e.g., design stage of software life cycle);

- plan the incremental transformation, integration, improvement, installation, development, and mapping of the software identified and specified earlier into the new or improved system (e.g., design stage of software life cycle);

- salvage and transfer through software improvement the retained existing software (e.g., programming stage of software life cycle);

- integrate existing external software packages by replacing existing internal software (e.g., programming stage and validation stage of software life cycle);

- build and test the new software components (e.g., programming stage and validation stage of software life cycle);

- test the new or improved system (e.g., validation stage of software life cycle);

- implement the new or improved system into production (e.g., operations stage of software life cycle); and

- review and evaluate the software improvement approach, processes, and results (e.g., review stage of software life cycle).

In summary, four basic advantages of the software improvement approach to improving or building a system over the conventional system development approach are that it-

- minimizes uncertainty and risk by maximizing the utilization of testable components;

- preserves the value of past software investment as much as possible and avoids the dangers of failure inherent in the "tear-it-down-and-start-anew" approach;

- enables the project to be broken into small, manageable pieces with an operational system at each phase; and

- is iterative in nature, which allows for the tasks performed to be repeated in an orderly, incremental fashion with constant achievement, growth, and feedback, until the overall objectives of the project are met.

2.1.3 Increments and Releases

Because of the large amount of software that exists in most ADP organizations, the software can't be improved in one "lump sum." Thus, the software is divided into smaller, more manageable groupings, called increments, that progress through the software improvement process as a work unit. Increments can be based on system, subsystem, or project boundaries, or by functional areas (e.g., input, edit, file update, report generation, or error handling). The key is to subdivide the software minimizing the interfaces between the groupings. The absence, or minimization, of

increment interfaces makes the improvements for each increment more independent, and allows the concurrent improvement of several increments at a time.

Also, because of the vast differences that may exist between the current and desired software environments, needed improvements can't be accomplished in one "quantum leap." Thus, the improvements for each increment are accomplished in multiple steps, called releases [3], consisting of logical sets of improvement activities that can be performed at one time. Improvements are normally subdivided by activity type into the three basic releases of-

- conversion;
- refinement; and
- enhancement.

Under these three releases, the types of improvement activities range from simple translation of code to complete re-engineering of existing systems.

Conversion-type improvement activities transform the software, without functional change, standardizing it and making it environment independent. Without standardization and independence, the next two releases, refinement and enhancement, would be extremely difficult, if not impossible, to accomplish. Standardized, independent software lends itself to manipulation by automated means and proceduralized processes, and facilitates flexibility for future requirements (e.g., moving to a new environment). The major conversion-type improvement activities are-

- standardizing code, data names, and format;
- upgrading languages, dialects, and operating systems;
- translating low- to high-level languages; and
- removing architectural or environmental dependencies.

Refinement-type improvement activities modernize the software to a state-of-the-art status and improve software maintainability and programmer productivity. Refinement is a prerequisite for software enhancement to ensure enhancements are not being made to software with obsolete coding features (e.g., EXAMINE or ALTER statements in COBOL), or outdated or incorrect functional requirements. The major refinement-type improvement activities are-

- restructuring code;
- realigning code;
- removing archaic features;
- "cleaning-up" code;
- streamlining job streams;
- isolating system functions and data;
- modularizing code and input/output; and
- creating "retrospective" documentation.

Enhancement-type improvement activities optimize the value, quality, efficiency, and effectiveness of the software enabling easier technical redesigns, easier addition of modern "technological" features and capabilities, and more efficient and effective use of resources. Without enhancement, the standardized and modernized software may still not function efficiently or effectively, or fulfill the user's desired requirements. The major enhancement-type improvement activities are-

- enabling interchangeability of functions;
- performing technical redesigns;
- revising file organizations and accessing mechanisms; and
- adding potential for future state-of-the-art features.

The improvement activities flow from one release to the next; thus, there is no "clear cut" dividing line between each release, and some functional overlap is inevitable. Exhibit 2 illustrates a typical software improvement release flow, including release inputs and results. Exhibit 3 illustrates the typical activities associated with each release, and the possible functional overlap of some of the release's improvement activities.

Improvement activities do not have to be subdivided into these three basic releases or follow the suggested release flow. They can be combined into one large release, or further subdivided into multiple mini-releases. The decisions as to the number of releases necessary to improve each increment, and the improvement activities to be performed in each release, are dependent on the size of the increment, overall number and type of improvements required, and priority of the improvements.

2.1.4 Software Improvement Process Overview

The software improvement process consists of the four major phases of-

- Planning and Analysis;
- Preparation;
- Improvement; and
- Implementation.

Exhibit 4 illustrates, in a chronological order, the four phases, and optional SIP pilot project, of the software improvement process including each phase's major tasks and the possible sequence and overlap of the software improvement process phases. It must be stressed that the order in which the tasks are presented does not imply a sequential process. In fact, most of the tasks associated with each software improvement process phase are performed concurrently and in a phased manner (see Exhibits 3 and 4). The tasks are listed under the phase in which they require the most attention.

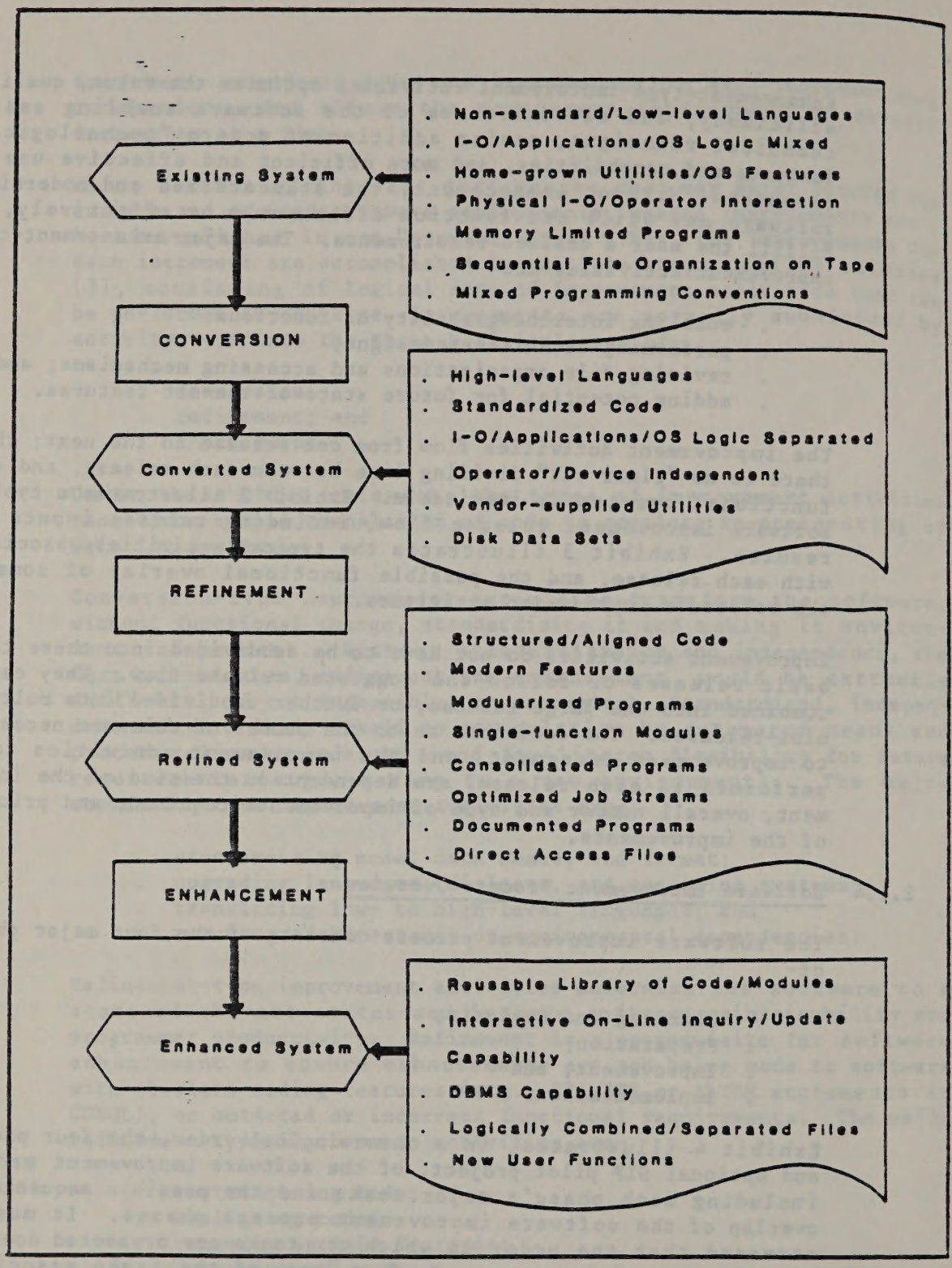


EXHIBIT 2: Typical Software Improvement Release Flow

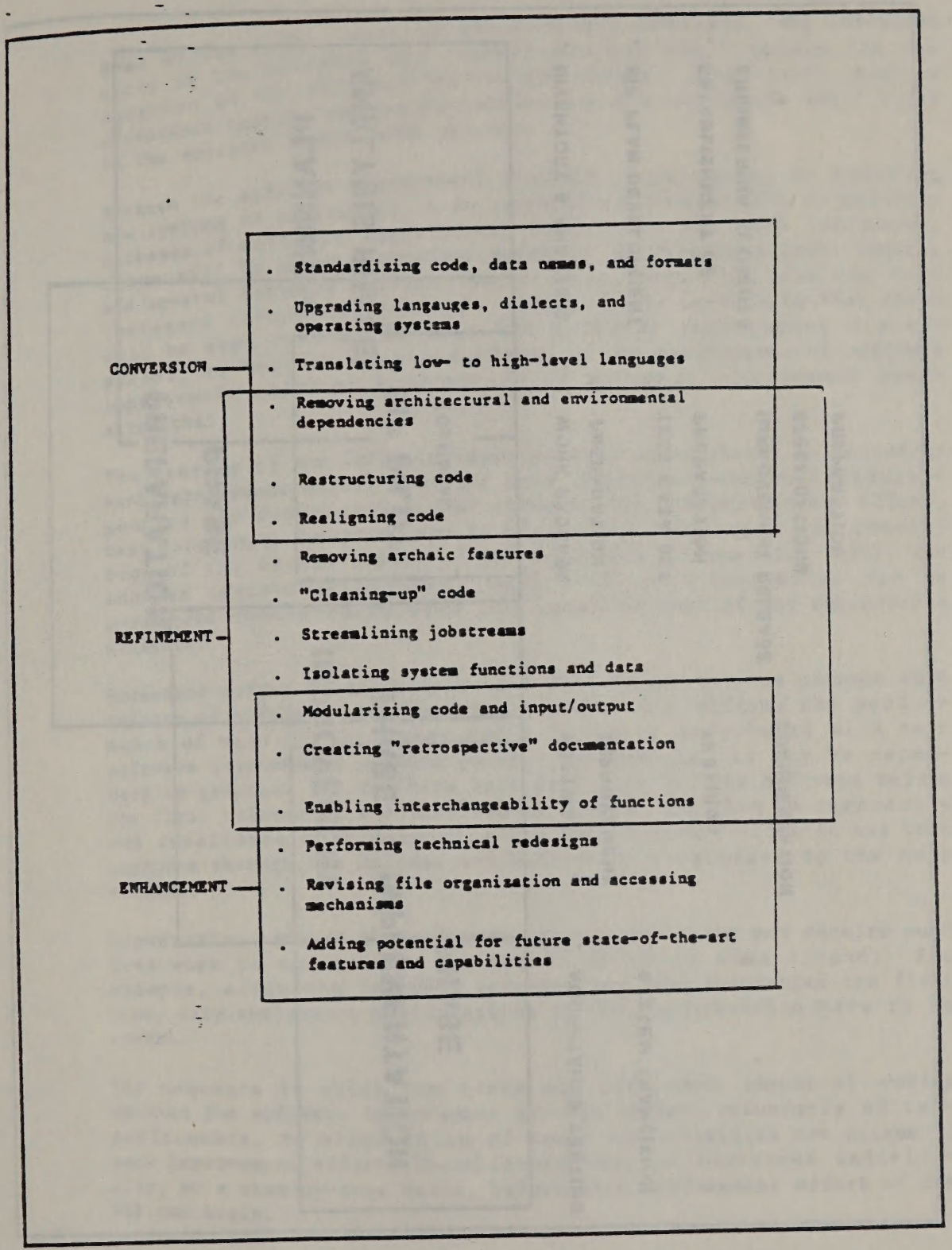


EXHIBIT 3: Software Improvement Activities Associated with Each Release

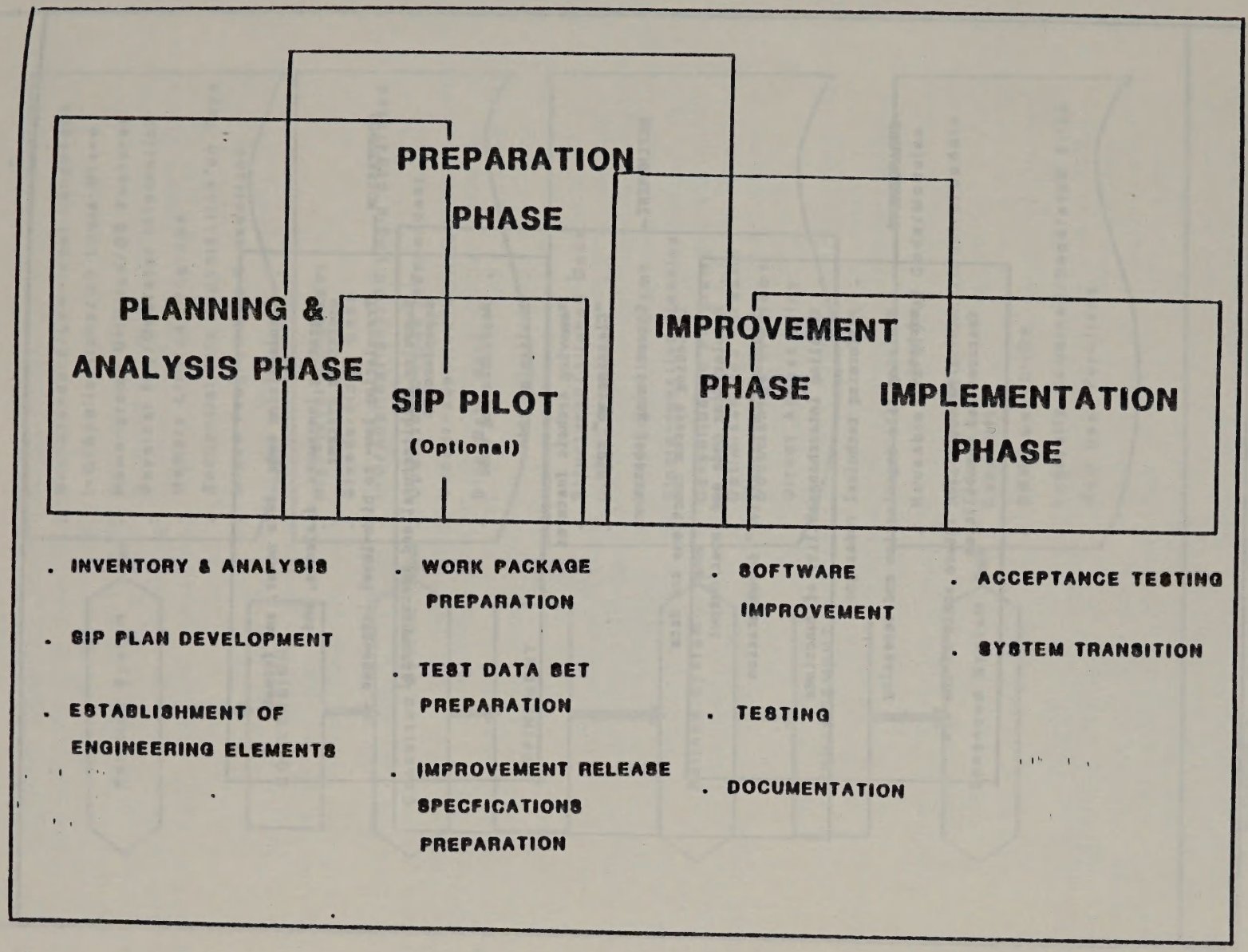


EXHIBIT 4: Software Improvement Process with Typical Phase Sequence

Some of the tasks, such as planning and analysis, are addressed early in the software improvement process and continue for the duration of the software improvement effort. Other tasks, such as acceptance testing and documentation, are not applicable until later in the software improvement process.

Because the software improvement approach to improving or building new systems is undertaken in an incremental fashion, and in multiple releases of software improvement activities for each increment, there will be many concurrent software improvement efforts ongoing and several software improvement release specifications for each increment throughout the SIP. Thus, it is inevitable that there will be some overlap between the software improvement process phases, as well as some redundancy in the performance of software improvement tasks or elimination of software improvement tasks altogether.

The overlap of the software improvement process phases is caused by each improvement effort's unique task sequence, and their contingencies and dependencies. For example, for one improvement effort, test data set preparation may be highly dependent upon the completion of the establishment of the engineering elements. While, for another improvement effort, test data set preparation can be performed concurrently with the establishment of the engineering elements.

Redundant software improvement task performance happens because each release of software improvement specifications requires the performance of many of the tasks and activities associated with each software improvement process phase. For example, it may be necessary to generate and validate test data sets for the software before the first release of the improvement effort, and then to regenerate and revalidate test data sets for the software after it has been improved through one release and before it progresses to the next release.

Conversely, some of the software improvement tasks may require much less work to accomplish the second or third time around. For example, after the improved software has been documented the first time, only changes or modifications to the documentation have to be posted.

The sequence in which the tasks are performed, amount of overlap between the software improvement process phases, redundancy of task performance, or elimination of tasks and activities are unique to each improvement effort. These issues must be addressed individually, on a case-by-case basis, before each improvement effort of the SIP can begin.

a. Planning and Analysis Phase

The Planning and Analysis phase embraces the three major tasks of-

- performing a software inventory and analysis;
- developing a SIP plan; and
- establishing engineering elements.

As part of the inventory and analysis task, software and file inventories need to be collected, summarized, and validated. The inventories support activities required for preparing a SIP plan and inventory control functions essential during the software improvement process. In conjunction with the inventory activities, software analysis includes a hard assessment of the software status and its disposition alternatives (i.e., purge, leave alone, replace, improve, or redesign/newly develop).

The activities required to develop a SIP plan include establishing and developing improvement objectives, determining the improvement methodology, preparing task assignments, and formulating a written SIP plan. The activities needed to establish program-unique, and possibly organization-wide, engineering elements (i.e., the baseline SET) includes developing, analyzing, evaluating, selecting, and implementing standards and guidelines, procedures, tools, quality assurance mechanisms, and training.

b. Preparation Phase

The Preparation phase includes the tasks of-

- preparing work packages;
- preparing test data sets; and
- developing improvement release specifications.

Work package preparation consists of defining work package standards; identifying all programs, files, job streams, documentation, and test data sets to be included in each work package; physically assembling all work packages and their individual components; and establishing an inventory and control system for work package monitoring and tracking. Test data set preparation includes developing test plans, creating test cases and scenarios, and generating and validating test data sets to verify successful software improvements. The activities associated with software improvement release specification development includes describing the specifications; identifying the requirements to be included in the specifications; and developing specification format, content, acceptance criteria, and general guidelines.

c. Improvement Phase

The Improvement Phase covers the tasks of-

- improving the software;
- testing the improved software; and
- documenting the software.

Improving the software involves software, job stream, and file conversion, refinement, and/or enhancement in a multi-phased, incremental fashion. Testing includes unit and system testing the improved software using test data sets prepared in the second phase of the improvement effort. Comparing system test results against predetermined test results determines when the improved software can proceed to acceptance testing. Documenting the software involves creating retrospective documentation for software that has no documentation or for which the documentation is out-of-date, as well as updating existing documentation to reflect changes made to the software and files during the software improvement process.

d. Implementation Phase

Implementation is the final phase of the software improvement process, and includes the tasks of-

- acceptance testing; and
- system transition.

Acceptance testing involves validating improved programs, job streams, and operating instructions and procedures, as well as revised documentation, data files, and data bases, against the software improvement requirements. Results from the execution of the improved software in the target environment should duplicate results from the execution of parallel or previous runs in the source environment. Acceptable comparison of outputs from the source and target environments determines when to start production of the improved software in the new environment. System transition works in conjunction with acceptance testing, and controls how the software will be migrated to the production environment through complete parallel operations, immediate transition, or phased parallel operations.

2.2 Major Goals of a SIP

As stated earlier, there are many goals for a SIP to achieve. The most important of them being to-

- improve software maintenance and control;
- reduce delays in responding to user's needs;
- improve software quality;
- increase programmer productivity;
- decrease software maintenance costs;
- institutionalize processes;
- change software from a reactive to proactive state;
- extend the software's life; and
- put the organization in a position to take advantage of new and emerging technology.

However, the end goals of a SIP are not only to improve software maintenance and control, but also to achieve as much isolation of function and standardization of interfaces within the software systems as possible. The achievement of these goals is attained through-

- isolating system functions;
- allowing for interchangeability of system functions; and
- facilitating change of elements within function.

Isolating system functions through modularization is a natural step towards avoiding reliance upon one architecture or environment, and increasing software maintainability and understandability. As functions are isolated, more design alternatives are presented and further possibilities of segmentation emerge. Thus, isolation of function holds the key to selecting cost-effective and efficient system alternatives in the future.

Functions should also be interchangeable with alternative design realizations to facilitate functional interfaces. This interchangeability of function, usually achieved through the use of reuseable and standardized modules of code, ensures an easier change in the means of performing a function (e.g., exchanging a called module that accesses a tape file for a called module that accesses a disk data set).

Facilitating the change of elements within functions refers to the software's portability and maintainability. Better portability and easier maintainability through the use of single-function, standardized, and reuseable modules of code is paramount to achieving the goals of a SIP. Easier changeability of the software, and its functions, increases and ensures more efficient use of the key data

processing-resources, especially people and machines. Standardizing, modularizing, parameterizing, and documenting the software are several techniques that facilitate the change of elements within functions.

Improving the software's "quality" (i.e., making the software "better"), is probably the best available means of achieving the SIP's goals. Software quality is a measure of its excellence, worth, or value against some ideal or standard. Although quality is an ill-defined term, there are many specific properties, or attributes, by which it can be defined or measured [4]. Exhibit 5 illustrates a proposed hierarchy of the major software quality attributes and their subordinate attributes. Although the subattributes are listed under only one major attribute, it must be stressed that several of them could conceivably be listed under more than one major attribute. For the sake of clarity and to minimize misunderstandings, each subattribute has been listed only once, under the major attribute with which it is most often associated.

It must be noted that it is rarely possible for all software quality attributes or subattributes to be implemented. It is necessary to first define the SIP's goals, and then the improvement objectives for each individual software application. Appropriate trade-off decisions must then be made among the various quality attributes and subattributes, as well as the goals and objectives to be achieved. For example, some processing efficiency may have to be sacrificed to achieve more maintainability, and vice versa.

The primary attribute all software is expected to have is useability [5]. Useability is the extent to which the software is reliable, efficient, portable, and maintainable. Without useability, the software is worthless, and it's remaining quality attributes are meaningless.

a. Reliability

Reliability is the extent to which the software correctly and satisfactorily performs its intended functions [5, 6]. It implies the software is dependable, accurate, and implementable. Dependability is the extent to which software can be relied upon to consistently function in a specified manner at specific times. Accuracy is the degree to which software produces results that are sufficiently precise to satisfy their intended use. Implementability is the extent to which, and ease with which, the software is able to be put into production or operation.

b. Efficiency

Software is efficient if it economically performs its functions and fulfills its purpose without waste of resources [6]. Resources include such items as funds,

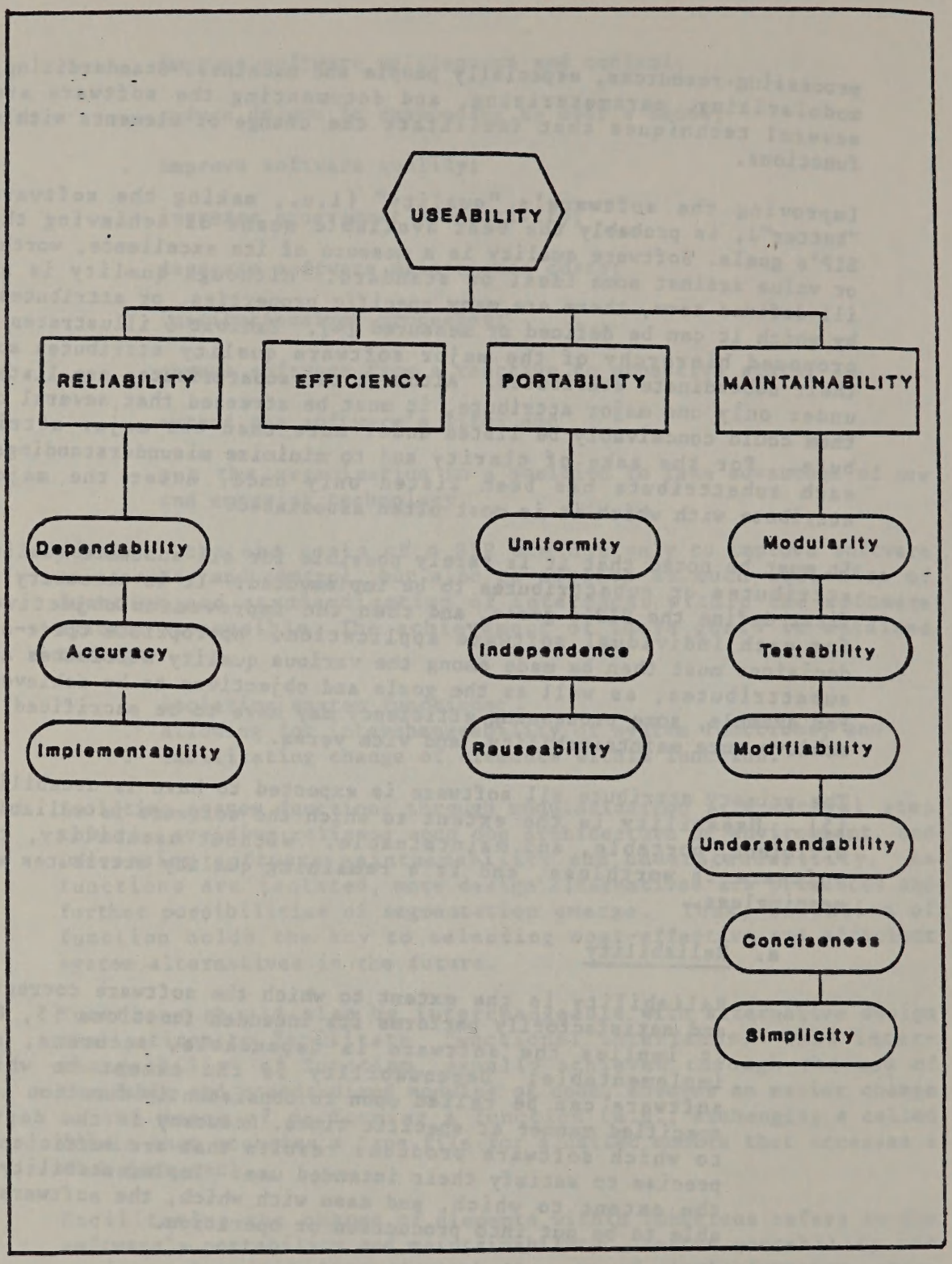


EXHIBIT 5: Hierarchy of Software Quality Attributes

time, personnel, computer time, main memory, communication channel capacity, and materials. Efficiency can be achieved by optimizing software processing time or storage, or economizing on costs and time. Optimizing processing time implies that alternative coding constructs are selected to produce more efficient object code. Optimizing storage implies that the data is packed where memory usage is high. Economizing on costs or time includes personnel costs and time involved in software engineering activities, as well as ADPE usage costs and time. However, it should be noted that making software processing efficient is not necessarily cost-effective in the long run, especially considering the need for more long-term and long-lasting quality attributes such as maintainability and portability.

c. Portability

Software is portable if it can be moved to, and operated easily on, other computer configurations and operating environments [5]. Portability implies the existence, to some degree, of software uniformity, independence, and reusability. Software uniformity is the existence of standardization such as naming conventions, code alignment, and common formats and interfaces. Software independence is the degree to which the code is free of architecture, device, or environment dependencies and vendor compiler extensions. Reuseability is the ability of the software to be used over and over again for various situations or reasons.

d. Maintainability

Software is maintainable if it facilitates updating to satisfy new requirements, rectify deficiencies, or correct errors [6]. It implies that the code is, to some degree, understandable, testable, modifiable, modular, concise, and simplistic. Maintainability protects software reliability by supporting testable changes, and prolongs system life by supporting adaptation to new requirements and environments [5]. However, because maintainable software introduces some processing and storage overhead, it is not necessarily processing or storage efficient.

Code, and its documentation, is understandable if its purpose or function is easily discerned by the reader. Understandability implies that variable names or symbols are used consistently and uniformly, modules of code are self-descriptive, mnemonic variable names and parentheses are used even if not necessary to the function of the code, and the control structure is simplified or in accordance

with a prescribed standard. More importantly, data names should reflect the use of the data (e.g., use "TAX" as a data name instead of "X").

Testability reflects the ease with which software changes can be demonstrated, in a quantitative manner, to be correct [5, 6]. It facilitates the establishment of validation and verification criteria, and supports evaluation and measurement of its performance. This implies that requirements are matched (i.e., synchronized) to specific modules, and that diagnostic capabilities are provided.

Software is modifiable if it is able to be changed or revised for various reasons or purposes with relative ease. Modifiability implies that the code is either flexible or general. Flexibility is the ease with which the software can be changed or revised [5], such as adding another transaction type or making a correction to the code. Generality is the extent to which the software can be used for a variety of changing functions without introducing revisions [5]. Examples of general code are common subroutines for data base calls or standardized input and output modules.

Software is modular if it is built in small, manageable pieces of code that are independent and self-contained. Modularity implies the software is structured in such a way that there is only one entry and exit point to each software module, variables are "visible" only in the module in which they are used, and the inputs/outputs and software functions are isolated.

Software is concise if the amount of code necessary to perform the desired function is minimized. Conciseness may be efficient for storage, but may be inefficient for portability or simplicity purposes.

Simplicity is a measure of the degree of complex decision making present in a piece of code. It is a function of the number of possible execution paths and the control structures and variables used to direct path selection [5].

2.3 Major Benefits of a SIP

A SIP is labor, management, machine-resource, and, possibly, deadline intensive. However, in spite of the problems that will inevitably arise, a SIP can be successfully engineered and prove highly beneficial to the organization. The advantage of utilizing state-of-the-art technological advances, such as teleprocessing, data base management systems (DBMS), and mass storage, is one such

benefit. Also, a SIP provides the capability to use this modern technology without being "locked in" to architectural or environmental dependencies.

Another benefit is the potential for more efficient and effective programmer productivity. Existing software, after improvement, can be maintained much more efficiently and the programmer's span of control should be greatly increased. That is, after software improvement, a programmer can maintain significantly more lines of code or system functions due to the increased maintainability and understandability of the improved software. The result is increased availability of an organization's most scarce resource -- skilled programmers.

A SIP more efficiently uses key resources, both people and machines. More readily available junior personnel can be used for both new development and maintenance, with improved productivity, lower risk, and less training. The more senior personnel can be used for more advanced tasks such as systems design or analysis, or tool evaluation and selection.

Additionally, the incorporation of a SET, consisting of a synchronized set of software standards and guidelines, procedures, tools, quality assurance, and training implemented through and coupled with a dynamic, ongoing SIP, simplifies the learning required of programmers and analysts. The simplified learning enables the institutionalization of a single training program for a common methodology and consolidated goals and objectives.

More efficient use of the ADPE is also possible because the state-of-the-art ADPE will not simply emulate obsolete or out-of-date functions. Rather, it will perform the technologically advanced activities for which it was designed.

Many additional benefits can be achieved with a thorough, comprehensive, and well-planned, -analyzed, and -managed SIP. Some of these include-

- improved user service levels;
- more flexibility for future requirements;
- the capability for automatic documentation and/or code generation;
- enhanced error recovery, system debugging, testing, data integrity, and security features;
- increased software quality (i.e., reliability, efficiency, portability, and/or maintainability);

- improved quality of software end-products (e.g., reports, statistics, and programs); and

- a synchronized, formalized, and tested SET for the SIP.

3. NEED FOR SOFTWARE IMPROVEMENT PROGRAM (SIP) PLANNING

For all work involving software, there is a great temptation to "forget planning and start doing." This is true of both the technical staff, who dislike "paperwork," and management, who want "results" not plans. It is, of course, possible to plan too much, but it would be even more dangerous to not plan enough.

The planner's role is analogous to that of an architect or engineer. The investors want to see results and the tradesmen want to start work. Yet it is obvious, to construct a structurally sound building, which will require minimum maintenance and stand for many years, requires thorough planning. Similarly, the construction of a structurally sound software system needs the same thorough planning. Thus, the SIP plan can be viewed as the "blueprints" for the SIP and the resulting software.

Until recently, most attempts to upgrade or modernize existing software, and software maintenance and production practices, have been largely unsuccessful [7]. Some of the main factors contributing to these failures are that in many cases-

- there was no overall plan or structured approach developed;
- the problems to be solved were not thoroughly identified or studied;
- the alternative solutions were not thoroughly analyzed or compared, and sometimes generic solutions were assumed to be sufficient; and
- the modernization attempts focused primarily on the "end result" instead of the "transition to the end result," thus initiating software redesign or new development programs instead of improvement programs.

A comprehensive SIP plan is necessary to minimize the SIP's risk of failure, establish a framework for subsequent SIP direction and management controls, and ensure practical solutions to real problems. The SIP plan addresses the implementation of these solutions via easily understandable and practicable improvement guidelines with synchronized SET elements [2]. Furthermore, the plan must-

- identify where the ADP organization is "today" (i.e., state the problems and the status quo);
- identify where the ADP organization wants to be "tommorrow" (i.e., state the organizational objectives); and
- address how the ADP organization plans to resolve the problems, or otherwise improve upon the status quo (i.e., provide a step-by-step approach or methodology on how to implement the improvements).

Planning resistance is a common problem. Some typical excuses for not planning are:

- "I'm too busy. I don't have the time. I need results -- now."
- "Planning is too complicated. I'm not a planning expert and don't want to be."
- "What's the point of planning? Plans aren't accurate enough and always have to be changed."
- "Why should I plan? I'm doing fine without them so far."

The last excuse seems to sum up the issue -- why plan? The reasons and motivation for planning are many. Planning forces people to look to the future. With the rapid pace of technological change today, there is a premium on a coherent approach to implementing, managing, and controlling a SIP. Because of the many mitigating factors, such as tools, software languages, ADPE, methodologies, and standards and procedures, there is a definite need to avoid proliferations and incompatibilities. And, finally, because of the amount of time and money devoted to the SIP, there is the need to get the fullest potential benefits from the SIP investment.

Thus, the single, most important factor of a SIP's success is not in establishing the best SET, having the lowest costs, achieving the fastest return on investment, or implementing the best improvements, but rather instituting the best SIP plans and management controls possible. Remember, any plan is better than no plan. However, "plans" themselves are nothing; "planning," and the execution of the plans, are everything.

3.1 SIP Plan Purpose, Content, and Scope of Work

A SIP plan serves as the program's single, master source of information. It provides direction for orderly software growth and change, minimizes the impact of information and technological change, and acts as a baseline for all program activities and schedules. Because many SIP's encompass vast amounts of software, many varied projects, and multiple organizational entities, it is difficult, if not impossible, for one plan to be "all inclusive" (i.e., provide enough detail and direction for all aspects, requirements, and levels of the SIP). Thus, several plans, or better yet, a hierarchy of plans at both the macro- and microlevels, is needed to adequately define the scope of the SIP and describe in detail the specific software improvement requirements and methodologies.

While addressing, in a general way, the overall need for, requirements of, and strategies for the SIP, macroplans-

- impose boundaries on subsequent SIP microplans;

- describe, in as much detail as possible, the SIP's increments, and, if possible, the increments' releases;

- describe and define the applicable software improvement process phases, tasks, and activities required to implement the SIP; and

- present common or generally applicable goals and objectives, problems and planning considerations, assumptions and constraints, and software improvement strategies and philosophies.

A macroplan should be updated and revised periodically, as significant program milestones are reached, especially at the end of each improvement effort (i.e., after each increment's release is completed, or after the improvements for an increment are finished).

Conversely, microplans are smaller in scope, but considerably more detailed than a macroplan. Microplans are required for each SIP increment, and provide specific step-by-step instructions on how to accomplish the increment's improvements. They designate specific task assignments and responsibilities, and specify definitive schedules and milestones. While operating within the bounds set by the macroplan, microplans-

- describe, in as much detail as possible, the SIP increment and its releases;

- identify any exceptions, unique features or technicalities, or additions to the macroplan components; and

- document, as software improvement release specifications, the specific improvements required for each individual system, subsystem, program, module, job stream, and file.

Regardless of whether the SIP plan is at a macro- or microlevel, it should provide strategic direction to the SIP so the program doesn't stray from its major goals and objectives. SIP plans are dynamic documents that, once developed, need to be continually updated to reflect major decisions, accomplishments, and occurrences. They require periodic review and revision, particularly after each major improvement step, to reflect changes in direction, results of studies or analyses performed, management or organizational changes, requirement modifications, legislative mandates, and state-of-the-art technological advances.

A thorough SIP plan contains [8, 9, 10]-

- an identification and definition of the SIP objectives as they relate to the overall agency mission, user needs, and ADP organization;

- an identification and description of SIP tasks;
- a description of the selected software improvement methodology;
- a description of the ADP organizational structure and the general responsibilities of the SIP task force and team(s), including both in-house and contractor personnel;
- a description of the features that should be preserved and those that should be changed or added during the improvement;
- a description of the system functions that should be isolated or made interchangeable;
- a description of the software functions that are candidates for replacement by an existing package;
- a description of the user-defined symptoms, and root problems to be solved, as a result of the SIP, as well as a list of the major concerns from users and managers, including political, technical, and financial concerns;
- a description of the SIP's workload-sequence priority scheme and the problem-solution priority scheme, as well as the rationale for deriving these priorities;
- an identification of SIP's risks and an assessment of their impact upon the agency mission, user needs, and ADP organization with a description of any contingency or fallback plans;
- an identification of resources (i.e., personnel, ADPE, space, supplies, etc.) required for, or allocated to, the SIP;
- a tentative schedule of the SIP, including incremental releases, software priorities, resource utilization, milestones, and time estimates;
- a description of the software improvement release mechanisms including specification content, format, standards of performance, acceptance criteria, and controls;
- a description of a functional baseline, including the operational concepts for the source and target environment hardware, software, files, test data, and documentation;

- the establishment and description of all major procedures needed during the SIP, such as SET coordination and modification procedures, configuration and change control, work package turnover, improvement steps, testing, and software improvement release specification preparation;

- a definition and description of SIP standards and guidelines, including descriptions of the mechanisms or techniques used to implement them and measure their adherence;

- a description of any SIP pilot project or prototype that empirically demonstrates the success and feasibility of the improvements and the results desired;

- a periodic synopsis of significant or major occurrences or accomplishments during the SIP; and

- a definition and description of the quality assurance criteria and techniques to be used to measure the SIP's success, such as software maintainability, personnel productivity, schedule impact, transparency to users, disruption of services, and system performance and reliability.

The SIP plan should culminate in a viable, organized approach to the overall software improvement process. The scope of work for the SIP plan must be defined well in advance of other tasks and should encompass-

- the four phases of the overall software improvement process;

- all tasks required for the software improvement process; and

- the integration of the SIP with such other ongoing organizational projects as hardware and software procurements, daily software maintenance and operations, and new development.

The use of the term "plan" in the preceeding discussion is generic. Actually, a "set" of hierarchical plans is required for a SIP, especially for one of a sizable magnitude. In addition to the overall SIP macro- and microplans, several other subordinate plans, or subplans, also at macro- and microlevels, may be needed. Some of the areas these subplans may address include personnel, training, configuration management, documentation, testing, resource management, system transition, security and privacy, release specification control, project management and review, and procurement. Collectively, this set of hierarchical plans represents a total SIP plan [6].

During the Planning and Analysis phase of the software improvement process, plans are normally incomplete and general in nature. However, formulating strategies early in the SIP life cycle facilitates the planning of specific tasks later in the software improvement process. Throughout the duration of the SIP, plans are periodically refined until they are eventually formalized as specific, detailed work plans and schedules. These plans need not be long or detailed. Short, concise plans may suffice, as long as the details are covered to the extent necessary. Furthermore, depending upon the size and complexity of the SIP, the subplans may be included as sections of the overall SIP plan, integrated as appendices or separate volumes, or addressed separately as independent plans.

3.2 Top-Down, Incremental SIP Planning Methodology

Exhibit 6 illustrates a "waterfall" model for top-down, incremental SIP planning and implementation. The major features of this model are that-

- planning is performed in a hierarchical, stepwise fashion with higher-level plans (i.e., macroplans) influencing and/or controlling the lower-level plans (i.e., microplans and software improvement release specifications);

- each improvement increment and release culminates in a review and analysis activity whose main objectives are to provide feedback to the original macroplans and microplans, and strategic direction to subsequent improvement increments and releases; and

- as much as possible, earlier increment and release improvement processes and achievements are repeated in subsequent increments and releases.

Hence, macroplan development for the overall SIP, progresses to the development of microplans for each improvement increment, and software improvement release specifications for each increment release. At the completion of each improvement increment, or release, feedback from that effort is used to update or revise the SIP macro- and microplans to reflect significant accomplishments or changes in direction. Also, an assessment of the results and methodologies used for each completed improvement effort (i.e., release or increment), is used as input to the next increment's microplan, or the next release's software improvement specifications. Ultimately, the successful completion of each improvement increment and release should correspond to macroplan and microplan milestones and measurable achievements of the SIP.

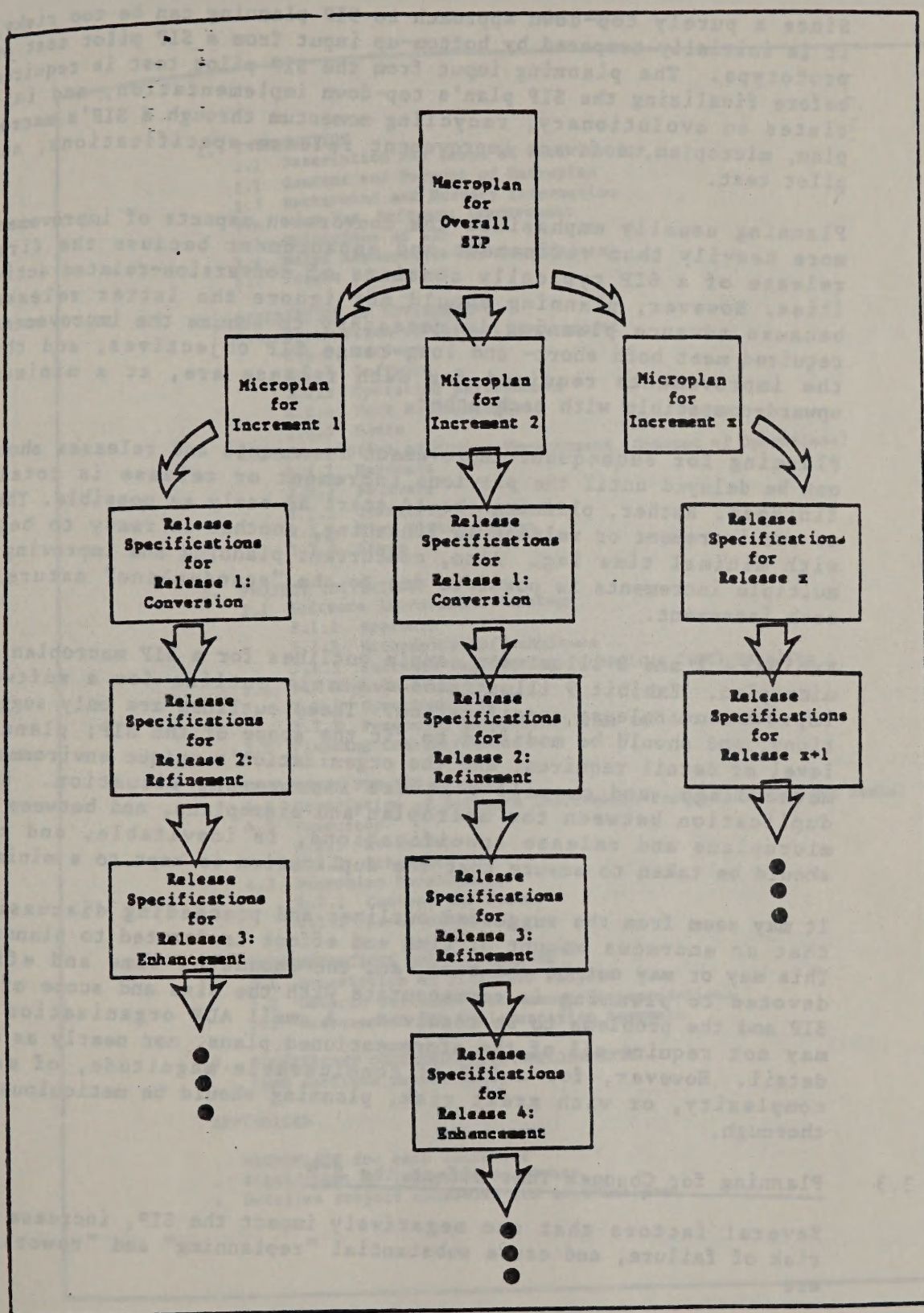


EXHIBIT 6: "Waterfall" Model for Incremental SIP Planning

Since a purely top-down approach to SIP planning can be too risky, it is initially tempered by bottom-up input from a SIP pilot test or prototype. The planning input from the SIP pilot test is required before finalizing the SIP plan's top-down implementation, and initiates an evolutionary, recycling momentum through a SIP's macroplan, microplan, software improvement release specifications, and pilot test.

Planning usually emphasizes the conversion aspects of improvement more heavily than refinement and enhancement because the first release of a SIP typically consists of conversion-related activities. However, planning should not ignore the latter releases because advance planning is necessary to ensure the improvements required meet both short- and long-range SIP objectives, and that the improvements required for each release are, at a minimum, upward-compatible with each other.

Planning for subsequent improvement increments and releases should not be delayed until the previous increment or release is totally finished. Rather, planning should start as early as possible. Thus, as one increment or release is finishing, another is ready to begin with minimal time lag. Also, concurrent planning and improving of multiple increments is possible due to the "stand-alone" nature of each increment.

Exhibits 7 and 8 illustrate sample outlines for a SIP macroplan and microplan. Exhibit 9 illustrates a sample outline for a software improvement release specification. These outlines are only suggestions, and should be modified to fit the scope of the SIP; planning level of detail required; and the organization's unique environment, methodology, and overall software improvement situation. Some duplication between the macroplan and microplans, and between the microplans and release specifications, is inevitable, and care should be taken to ensure that the duplication is kept to a minimum.

It may seem from the suggested outlines and preceeding discussion, that an enormous amount of time and effort is devoted to planning. This may or may not be the case, and the amount of time and effort devoted to planning is commensurate with the size and scope of the SIP and the problems to be resolved. A small ADP organization SIP may not require all of the aforementioned plans, nor nearly as much detail. However, for a SIP of considerable magnitude, of major complexity, or with great risk, planning should be meticulous and thorough.

3.3 Planning for Changes That Affect the SIP

Several factors that can negatively impact the SIP, increase it's risk of failure, and cause substantial "replanning" and "reworking" are-

1. INTRODUCTION
   - 1.1 Description and Scope of Work for SIP
   - 1.2 Content and Purpose of Macroplan
   - 1.3 Background and History Information
   - 1.4 Need for Software Improvement
   - 1.5 Objectives of SIP
   - 1.6 Major Assumptions and Constraints
   - 1.7 Points of Contact

2. DESCRIPTION OF ENVIRONMENT
   - 2.1 Description of Current Environment
     - 2.1.1 Hardware
     - 2.1.2 Software
     - 2.1.3 Operating Environment
     - 2.1.4 Work Methodologies
     - 2.1.5 Users
   - 2.2 Description of Future Environment (Concept of Operations)
     - 2.2.1 Hardware
     - 2.2.2 Software
     - 2.2.3 Operating Environment
     - 2.2.4 Work Methodologies
     - 2.2.5 Users

3. PROJECT INITIATION
   - 3.1 Software Improvement Strategy
     - 3.1.1 Approach
     - 3.1.2 Methodologies/Techniques
     - 3.1.3 Software Engineering Technology (SET) Baseline
   - 3.2 SIP Organization
     - 3.2.1 Structure
     - 3.2.2 Personnel Responsibilities and Authorities
   - 3.3 Planning Considerations

4. GUIDELINES FOR SIP PROCESS
   - 4.1 Description of Software Improvement Process (Phases and Tasks)
   - 4.2 Resources
     - 4.2.1 Estimates
     - 4.2.2 Schedules
   - 4.3 Microplan Development
     - 4.3.1 Content and Format
     - 4.3.2 Level of Detail

5. RECOMMENDATIONS AND CONCLUSIONS
   - 5.1 Description of Pilot SIP Project
   - 5.2 Generic Software Improvement Recommendations
   - 5.3 Recommended SIP Implementation Approach

6. SIGNIFICANT OCCURRENCES/ACCOMPLISHMENTS
   (Add sections periodically)

APPENDICES

- MICROPLANS for each increment
- Significant or important documents
- Detailed project schedules and cost analyses

EXHIBIT 7: Sample SIP Macroplan Outline

1. INTRODUCTION
   - 1.1 Description and Scope of Work for Increment
   - 1.2 Content and Purpose of Microplan
   - 1.3 Objectives of SIP for Increment
   - 1.4 Major Assumptions and Constraints
   - 1.5 Points of Contact

2. IMPROVEMENT STRATEGY FOR INCREMENT
   - 2.1 Description of Increment's Releases
   - 2.2 SIP Organizational Assignments
   - 2.3 Planning Considerations
   - 2.4 Modifications to Baseline SET

3. SOFTWARE IMPROVEMENT TASK ASSIGNMENTS FOR INCREMENT
   - 3.1 Task 1 - Inventory and Analysis
     - 3.1.1 Task Description
     - 3.1.2 Resource Estimates and Allocation
     - 3.1.3 Schedules

     (Repeat same components for 3.2 through 3.11)
   - 3.2 Task 2 - SIP Plan Development
   - 3.3 Task 3 - Establishment of Engineering Elements
   - 3.4 Task 4 - Work Package Preparation
   - 3.5 Task 5 - Test Data Set Preparation
   - 3.6 Task 6 - Improvement Release Specifications Preparation
   - 3.7 Task 7 - Software Improvement
   - 3.8 Task 8 - Testing
   - 3.9 Task 9 - Documentation
   - 3.10 Task 10 - Acceptance Testing
   - 3.11 Task 11 - System Transition
   - 3.12 Summary
     - 3.12.1 Summary of SIP Task Estimates
     - 3.12.2 Summary of SIP Task Schedules

4. SIGNIFICANT OCCURRENCES/ACCOMPLISHMENTS
   (Add sections periodically)

APPENDICES

- SOFTWARE IMPROVEMENT RELEASE SPECIFICATIONS for each release
- Detailed schedules and cost analyses for each release
- Significant or important documents

EXHIBIT 8: Sample SIP Microplan Outline

1. OVERVIEW OF RELEASE SPECIFICATION
   - 1.1 Description of Release
   - 1.2 Scope of Work for Release
   - 1.3 Objectives of Release Improvements
   - 1.4 Content of Release Specification

2. IMPROVEMENT STATEMENT OF WORK SUMMARIZATION FOR RELEASE
   - 2.1 Software Improvements Requirements
     - 2.1.1 Programs
     - 2.1.2 Files
     - 2.1.3 Job Streams
     - 2.1.4 Overall Processing
   - 2.2 Deliverables
   - 2.3 Standards of Performance
   - 2.4 Acceptance Criteria

3. SPECIFIC IMPROVEMENTS FOR RELEASE COMPONENTS
   - 3.1 System 1 - Title
     - 3.1.1 System Summarization
       - 3.1.1.1 System Description/Background
       - 3.1.1.2 Identification Section (Work Package, Date Prepared, Prepared By)
     - 3.1.2 Overview of Current Processing
       - 3.1.2.1 Inventory of Current Components
       - 3.1.2.2 Description of Current Processing
       - 3.1.2.3 Problem Summarization
     - 3.1.3 Overview of Revised/Improved Processing
       - 3.1.3.1 Inventory of Revised/Improved Components
       - 3.1.3.2 Description of Revised/Improved Processing
       - 3.1.3.3 Improvement Summarization (Goals)
     - 3.1.4 Detailed Component Improvements
       - 3.1.4.1 Program 1/Module 1
       - 3.1.4.2 Program 1/Module 2
       - 3.1.4.3 Program 2/Module 1
       - 3.1.4.4 Program 2/Module 2
       - 3.1.4.5 Program n/Module n
       - 3.1.4.6 File 1
       - 3.1.4.7 File 2
       - 3.1.4.8 File n
       - 3.1.4.9 Job Stream 1
       - 3.1.4.10 Job Stream 2
       - 3.1.4.11 Job Stream n
   - 3.2 System 2
   - 3.3 System n

EXHIBIT 9: Sample Software Improvement Release Specification Outline

- people;
- technology;
- physical constraints;
- costs;
- organizational objectives; and
- external conflicts.

All of these factors have one predominant commonality -- they cause "change." Changes to the SIP, and that can affect SIP plans, should try to be anticipated, and if possible, planned for in advance. Several ways in which changes can affect a SIP and it's planning are:

- A major organizational or system policy change can't be forestalled, but can substantially alter the organization's mission or the user's needs. This change, and subsequent alteration of mission and needs, can have a "domino effect" on the SIP objectives, estimates, schedules, and costs.

- The software being improved may require changes due to error correction or system enhancement. Some examples of software-changes are a change to the system's logic flow or input or output elements, formats, or media; a modification of system functions or data transactions; and a need for new or additional system features or controls. Software changes can be planned for and controlled by implementing appropriate software change control procedures during software improvement such as software version releases and maintenance freezes.

- Inventory changes, such as programs or files not previously found or now determined to be eliminated, can cause additional problems and substantial modifications to the SIP work packages, schedules, priorities, and resource estimates. If the improvements have been contracted out, these inventory changes may even require extensive contract "change orders" and large contract cost increases.

- Procurement schedule changes can "cripple", or even halt, the SIP if the improvements required are dependent on a new or modified software package, tool, operating system, or ADPE being procured and in place.

- The resources (i.e., time, people, ADPE, money, supplies, etc.) allocated to the SIP can also change. Sometimes system users or SIP managers are replaced, with their replacements causing changes to the SIP objectives, requirements, schedules, etc. Changes in resource availability (e.g., key personnel, sufficient ADPE capacity, computer time, or access to terminals) can substantially slow the SIP and require extensive schedule and priority modifications. These resource changes relate to the man-machine interfaces

required during a SIP and primarily impact the Improvement phase of the software improvement process. They can be partially controlled by employing techniques in ergonomics (i.e., human engineering) along with thorough SET and SIP planning, management, and coordination techniques.

- Major changes in the goals or priorities of the SIP may also occur. Some goals will be found to be next to impossible to achieve, some will conflict with each other, some will conflict with the constraints on the SIP, and some will be too costly, versus the benefits gained, to pursue. Some goals will be misunderstood and/or difficult to measure, some will please only a small audience (e.g., top management or technicians), and some areas for which goals should have been set will be overlooked. Additionally, some of the priorities will be found to be misplaced, erroneous, or too political to handle.

3.4 Planning Considerations

A SIP typically present both technical and managerial problems such as software modularization techniques or organizational task assignments. Thus, a thorough, comprehensive technical and managerial approach (i.e., the SIP plan) must be in place to anticipate and avoid as many of these pitfalls as possible. The key to establishing a workable plan, which minimizes the occurrence and effect of technical and managerial problems on the SIP, is planning.

3.4.1 Planning Considerations for Technical Problems

Technical problems during a SIP are primarily due to-

- the thoroughness and level of detail required in the analysis of the software to identify and describe the improvements required;
- the existence of software, file, and system interrelationships and dependencies; and
- the fact that improvements are subjective and qualitative in nature (i.e., different for everybody and every situation).

The last point is particularly true because ADP state of the art is a moving target, and software improved today may become obsolete or outdated tommorrow. However, the improvements made today, due to the improved software quality and programmer productivity, should aid in implementing any future improvements or redesigns.

Performing a prototype or pilot project that empirically demonstrates the feasibility and success of the improvements and the methodology will help solve, early in the SIP life cycle, any technical problems that may occur. Prototyping the SIP is a most

effective strategy given that some of the improvement issues, objectives, requirements, methodologies, etc., will be ill-defined, unproven, and/or misunderstood. Prototyping is preferable to extensive, time-consuming, and, sometimes, highly theoretical "paper analysis." Pilots risk minimal investments of effort and money up front, while delivering benefits early, and provide for technology integration without locking the organization into one method.

It is imperative for the success of subsequent SIP planning that the prototype's software improvement methodology and results be recorded and evaluated, and any successes be widely advertised to gain support for the SIP. It is of utmost importance to a SIP to have successful early pilots. Dramatic returns on investments are possible, and successes are needed to initiate and encourage involvement of external SIP personnel, such as users and top management. To ensure that the pilot project is realistic and that early feedback is achieved, real production software should be used.

The key to a useful pilot test is rapid prototyping of the SIP with the "research and development" activities performed early in the pilot project's life cycle, thus avoiding the need for extensive R&D later in the project. The pilot project should not be implemented before a plan is developed due to the risk of technical chaos. Early, successful pilots will ease later SIP integration by focusing on established and tested SET and software improvement methodologies. Thus, most of the technical problems can be identified, analyzed, researched, resolved, and planned for, before the SIP continues on a larger scale.

3.4.2 Planning Considerations for Managerial Problems

SIP managerial problems are primarily due to-

- the voluminous amount of data to be managed and controlled because each program has its own associated source code, listings, test data, test cases, test scenarios, test results, documentation, and jobstreams;
- the different, multiple tasks to be performed and coordinated;
- the interfaces, dependencies, and concurrencies between some of the tasks and between the software being improved, controlled, and coordinated;
- the large number of physical entities to be identified, handled, monitored, and controlled; and
- the supervision and management required over the many and varied types of individuals involved with the improvement effort.

Because no single improvement effort can avoid all anticipated pitfalls, problems must be confronted early in the software improvement process to solve them, or at least cushion their impact on the SIP. To reduce the occurrence of problems during the software improvement process, continuous review of the SIP plans and monitoring of the SIP status are necessary. Additionally, constant coordination with top-level management and executive personnel may provide advance information allowing for the timely revision of plans and schedules, and the execution of contingency plans.

Numerous "headaches" are the result of unrealistic SIP expectations. Some examples of unrealistic expectations include overestimating programmer or tool productivity and capability, expecting no interruptions to everyday production or operating procedures, expecting to optimize the hardware or software efficiency or make major functional changes to the system while improvements are being made, and expecting the coding to start before the planning is finished. To avoid such expectations, the SIP goals and objectives should be established and documented as early as possible. If, afterwards, it is discovered that these contain some inconsistencies or errors, then the objectives should be amended. Also, if the improvement requirements are found to be too rigid or unrealistic, having the rationale for selecting the original improvement requirements documented in the SIP plan will aid in revising them.

In this same context, underestimating the role of the user, or the total time and resources required for the SIP, are situations that should be anticipated during contingency planning. Allow adequate time and resources for the completion of all software improvement tasks. Take staff capabilities as well as limitations into consideration when developing the SIP schedules and assigning tasks, anticipating some schedule slippage. Providing a "reserve" of personnel resources and time is an excellent means of affording flexibility to the schedules and plans.

If the improvements are to be contracted out, it may be advantageous to allocate extra time and resources for contract negotiations, change orders, and acceptance testing. Detailed planning of a request for proposals (RFP), in terms of expected deliverables, task performance, and acceptance criteria, is necessary well in advance of the contract advertisement and award to avoid future contractor problems.

Biased decision making and predetermined results can also hinder SIP planning. Biases or vested interests may result from organizational affiliations, friendships, politics, and personal viewpoints. Predetermined results can be caused by political pressure, having a vested interest in the SIP outcome, or inadequate research and analysis. Examples of these include developing project schedules based only upon a management prerogative, such as lowest cost or shortest time frame; concentrating improvements exclusively on only one or two areas of the software, those most preferred by or that

directly affect the planner; and inadequately researching topics or analyzing software because the results are assumed to be known, obvious, or commonly accepted. Planners need to determine, as early as possible, if the SIP personnel have any biases or predetermined concepts about the SIP, its objectives, or their tasks, particularly before making task assignments or developing detailed schedules. Conducting briefings and training sessions early in the SIP life cycle may help keep SIP personnel informed and avoid biases from forming.

Failure to allow for adequate lead time is another pitfall for which to plan. Allow adequate lead time to establish and implement a SIP organizational structure; staff the SIP with skilled and key personnel; complete a thorough, comprehensive inventory and analysis of the source and target environments; and plan and implement SIP training, allowing for a learning curve for the SIP personnel.

Often the people chosen to manage a SIP are not "great" managers, but rather "great" technicians or sociable people. This lack of management and planning expertise can result in an overly technical software improvement approach, without proper emphasis on managerial concerns, causing the SIP to go off course. The other extreme -- too general an approach -- is also undesirable, as it leaves the SIP task force and team(s) floundering without direction. Thus, it is best to staff the SIP with managers who have both high managerial and technical abilities. While the technical aspects of improvement, such as system design, coding, tool implementation, and conversion, are desirable, they are not the most important attributes a manager needs. Project planning, management, analytical and problem solving, conflict resolution, communication, and organizational abilities are some of the key skills required of effective managers.

Another area to consider is the actual project management. A lack of commitment to the SIP and its goals and objectives can be a big problem. A SIP of considerable size and complexity requires management, programmer, analyst, and user support; availability of adequate budgets; allocation of key personnel, computer time and capacity, and supplies; and definite assignments of responsibility and authority. As in any major project, conflicts are bound to occur concerning responsibilities, authorities, levels of authority, chains of command, and priorities. Remember, not everyone will always be happy with the decisions made; however, the manager's main objectives are to successfully achieve the goals of the SIP, while at the same time being as fair and consistent as possible in dealing with the SIP personnel.

SIP planners should also plan for morale problems and personnel turnover. These can occur in any project; however, a SIP is especially vulnerable because, except for the Planning and Analysis phase, it is a highly mechanical and repetitive process. Therefore, it takes away much of the innovation and experimentation usually

afforded to programmers and analysts. Highly visible and innovative technical areas, such as procuring hardware or selecting tools, often overshadow the more mundane tasks of a SIP, such as preparing software improvement release specifications or work packages. It must be stressed that the nature of a SIP is such that it demands that it be highly proceduralized and standardized to achieve results that are uniform, and within the constraints of the organization's long-range ADP plans, the user's requirements, and the SET.

It is inevitable that there will be some loss of personnel during a SIP due to transfers and terminations. Planners should expect, and plan for, some personnel turnover. Expanding SIP personnel responsibilities, and hence their self-esteem, by including them in planning sessions, asking for ideas for future enhancements or improvements, rotating or changing assignments and responsibilities, conducting periodic briefings on the project status, and providing adequate training can help to alleviate morale problems and personnel losses.

4. IMPLEMENTATION OF A SOFTWARE IMPROVEMENT PROGRAM (SIP)

4.1 SET and SIP Interrelationship

As previously discussed, the SIP works closely, and in tandem with a SET. A SET consists of a synchronized group of the five software engineering elements of-

- standards and guidelines;
- procedures;
- tools;
- quality assurance (QA); and
- training.

These five elements direct and control all software activities throughout the software's life cycle [2] and for different software engineering or re-engineering purposes (e.g., software development, maintenance, improvement, conversion, or redesign). Thus, the SET addresses, on an organization-wide basis, the software engineering methods, metrics, and latest controls for managing an installation's software activities, while the SIP addresses the upgrading of the existing software, job streams, and files with regard to the SET baseline.

The same five engineering elements of a SET are required, in a specialized sense, for a SIP. The establishment of these five engineering elements as a formalized SET is paramount to the successful implementation of a SIP. Improvement and installation standards, guidelines, and procedures are required to standardize the software activities and the software, so they can be measured, controlled, and improved. Specialized improvement tools are both necessary and desirable to increase programmer productivity, enforce systemization, and improve controls. QA is required to ensure that the improvements made actually resolve the problems, quantatively prove that the SIP is a viable and worthwhile effort, identify and measure the resultant improvements and benefits, and control and enforce the quality of the software and the improvement performance. Finally, training and retraining are also necessary, for without them successful accomplishment of the SIP would be next to impossible and the improved software and methodologies would quickly degrade.

The question then becomes a paradox similar to the "chicken and the egg" -- "Which comes first, the SET or the SIP?" The establishment of a SIP and a SET are separate, but interrelated and coordinated processes. Each can be established independently of the other, but each has a controlling or influencing effect on the other. That is, the standards and guidelines, procedures, tools, QA, and training established for the installation as a SET, define the framework for

and provide a baseline from which, the SIP can operate. In this sense, the SIP cannot, for example, set standards that oppose those instituted in the SET, or use tools that conflict with the tool's technology chosen for the organization and established in the SET.

Conversely, standards and guidelines, procedures, tools, QA, and training, when established in a SIP, limit the choices of the SET. For example, the SET cannot institute software or installation standards different from those just implemented by a SIP, or maintained and enforced by the improvement tools. If either the SIP or SET institute engineering elements without considering the organizational impact, or short- and long-term consequences, the resulting software and engineering activities will, at best, be chaotic and consist of a "patchwork" of styles, structure, and standards.

In determining which to establish and implement first, the SIP or the SET, the organization has three choices:

a. SIP First, Then SET

If the organization already has existing software, it can choose to build on their past investment in this software through a SIP, as opposed to the "big-bang theory" of redesign or new development. In this case, the organization would establish and implement a SIP with the engineering elements as program-unique components of the SIP. At some later point in time the program-unique engineering elements may be found to be applicable on an organization-wide basis, prove to be effective and efficient, or evolve into "de facto" organization-wide engineering elements. Then, they can be instituted into a formal, organization-wide SET. The end-result of implementing a SIP first, and then the SET, is an organization-wide SIP that experiments with, and implements on a small scale, program-unique engineering elements before instituting them on an organization-wide basis, or applying them to other engineering activities, and that retrofits existing software to the organization-wide SET.

The advantages of establishing the SIP first, and then the SET within it, are that it-

- allows for experimentation on a small scale, such as a SIP pilot project or one SIP increment, before formalizing and instituting them on a program- or organization-wide basis, or for other engineering activities (e.g., new development, redesign, and maintenance), thus controlling and influencing the organization-wide engineering elements; and

- reduces the scope and impact of error correction if inadequate or incorrect engineering elements are adopted.

The disadvantages of this alternative are that it-

- does not address the improvement of other ongoing engineering activities, such as software development or redesign, until some later point in time; and

- may result in a SET that is too narrow in scope and isn't applicable on an organization-wide basis.

b. SET First, Then SIP

If the organization does not have any existing software (i.e., a new ADP organization), or decides to ignore its existing software investment, then a large redesign or new development effort is necessary (i.e., the "tear-it-down-and-start-anew" approach). In this case, the organization would establish an organization-wide SET before starting the redesign or new development. It is paramount to the success of the redesign or new development activities that a SET be in place prior to redesign or new development. The establishment and use of a SET ensures the newly engineered, or re-engineered, software uses structured analysis, design, and coding techniques; modern programming practices; and formalized standards and procedures, so the resulting software will be well-documented; be easy to support, use, understand, modify, and enhance; fit the application better; and be less expensive and time consuming to maintain.

Once the high-quality, well-structured software has been delivered and is operational, QA mechanisms serving as a software preventive maintenance program must be established to ensure that the software does not degrade during subsequent software maintenance or conversion activities. A SIP is just such a QA mechanism for this preventive maintenance. The end-result of implementing a SET first, and then a SIP is an organization-wide SET that experiments with, and implements on a wider scale, organization-wide engineering elements before instituting them on an organization-wide basis, or applying them to all software engineering activities, and that is retrofitted into existing software through a SIP.

The advantages of establishing a SET, or at least a baseline SET, prior to a SIP, are that the-

- current organization-wide engineering elements, such as directives and guidelines, can be formally instituted, and help in determining the direction and actual specifics of the SIP; and
- organization-wide engineering elements can be implemented early for many different engineering activities (e.g., new development, redesign, improvement, and maintenance), and will constrain and influence the program-unique engineering elements.

The disadvantages of this alternative are that it-

- may result in a SET that is too rigid and not flexible for each application;
- may result in a SET that is too broad in scope and not applicable to each application; and
- may be too theoretical in its approach, and thus too difficult, if not impossible, to conform to the SET's standards or follow the SET's procedures during a SIP.

c. SIP and SET Together

An optimum, compromise solution is a "double-barreled" approach to resolving software problems -- a "software and technology modernization program" (STMP). Exhibit 10 illustrates a typical interrelationship of the SIP and SET as integral parts of a STMP.

The advantages of this alternative are that it-

- encompasses all ongoing software engineering activities (e.g., software development, redesign, improvement, and maintenance);
- is a flexible approach to the problems of being too rigid, or too broad or narrow in scope;
- allows for experimentation of the SET on a small scale; and
- reduces the scope and impact of error resolution.

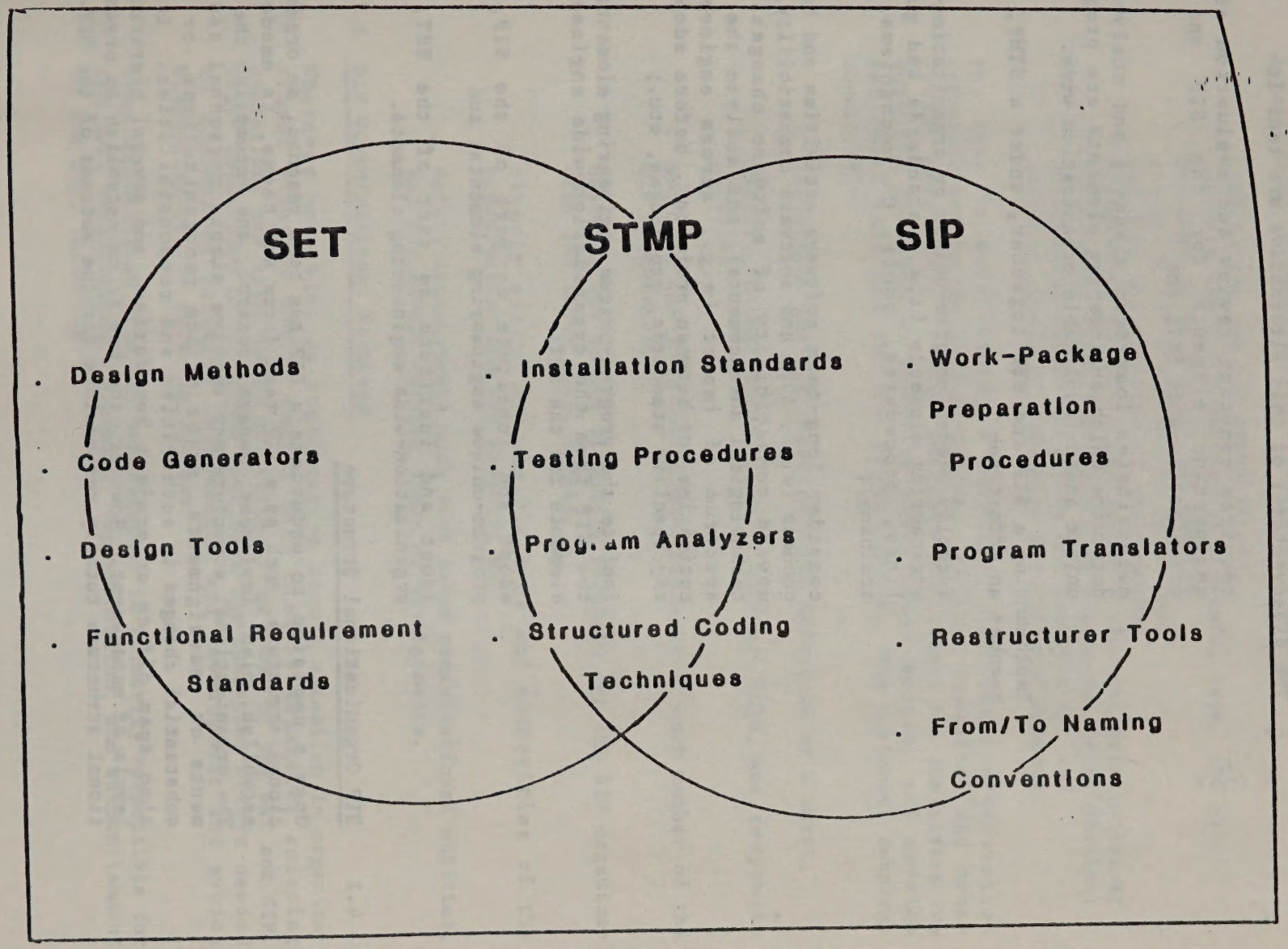


EXHIBIT 10: Example of Typical SET and SIP Interrelationship under a STMP

The disadvantages of this alternative are that it-

- requires constant review and evaluation of the engineering elements for the SIP and the organization-wide SET; and

- necessitates long-term planning and analysis to determine which engineering elements are program-unique and/or applicable organization wide.

Implementing a SIP and SET together, under a STMP, thus becomes an effort to-

- identify needed program-unique or organization-wide engineering elements (i.e., standards and guidelines, procedures, tools, QA mechanisms, and training);

- consider long-term software activities and consequences (e.g., ADPE and software compatibility, the upward compatibility of software changes, and technological advancements), and analyze the full spectrum of impact (e.g., across engineering activities, or between projects), before adopting any specific standards, procedures, etc.;

- isolate the program-unique engineering elements for the SIP from the organization-wide engineering elements for the SET;

- adopt and institute as part of the SIP, the program-unique engineering elements; and

- adopt and institute as part of the SET, the organization-wide engineering elements.

4.2 SIP Organizational Structure

Once a decision to undertake a SIP has been reached, an organizational structure, such as a SIP task force and teams, is needed to establish, plan, implement, manage, control, and accomplish the SIP. The organizational structure may involve simple personnel assignments or reassignments, major office reorganizations, or even substantial changes in authorities and responsibilities. It may also span office or project boundaries, and several hierarchical levels of management. However, the key is to establish an organizational structure totally responsible for the success of the SIP.

Establishing an organizational structure requires-

- instituting the SIP organization;
- defining the SIP organization's nature, type, and size;
- designating an overall SIP manager;
- assigning SIP organization personnel;
- defining the duration of assignments;
- documenting the SIP organization's internal relationships;
- documenting the SIP organization's external relationships;
- designating responsibilities and authorities; and
- defining levels of authority and chain of command.

The keys to establishing a workable, functional organization structure are to assign high-caliber, skilled technical and managerial personnel, provide adequate logistical and administrative support, allocate an adequate operational budget, and develop personnel subplans for the organization. The personnel subplans should [9]-

- illustrate the SIP organizational structure as a chart;
- explain the SIP organization's charter, role, and responsibilities;
- explain the role and responsibility of each member of the SIP organization;
- describe the number of people required for the SIP organization and their skill levels;
- clearly assign responsibilities and authorities of the organization and its members; and
- detail arrangements made with other organizational entities, and any subcontracting plans or arrangements.

4.2.1 SIP Organizational Strategy

Wherever possible, an organization should establish an organizational strategy and/or entity that clearly separates existing operations and maintenance activities from those of the SET and SIP. This "separation of functions" achieves the concerted effort needed to accomplish software improvement and development and avoids disruption of the existing systems.

Once established, the SIP organization is primarily responsible for the full establishment, planning, implementation, and accomplishment of the SIP, including-

- strategic planning and plan performance;
- configuration control and program management;
- the software engineering technology (SET);
- systems requirements, analysis, design, and development;
- testing the improved software; and
- software package specification development and acquisition.

This SIP organization, and any subordinate organizations it may have, can be viewed as a SIP task force. Once improvements are ready to begin, SIP teams will also be required to accomplish the improvements, or act as liaisons with, or technical representatives to, improvement contractors.

4.2.2 SIP Task Force and Team Structure

A SIP task force, and its subordinate teams, function akin to the human body with the task force acting as the "brain" (i.e., planning, controlling, and monitoring the functions of its appendages), and the teams acting as the "arms and legs" (i.e., carrying out the brain's orders). The SIP task force is the focal point for all SIP planning, and functions as the nucleus of the SIP by-

- initiating the SIP;
- developing SIP plans;
- directing, controlling, and managing the SIP;
- gathering data and researching software improvement areas;
- performing studies and analyses;
- establishing SIP objectives;
- resolving major technical and managerial problems;
- resolving conflicts and discrepancies;
- establishing improvement schedules and resource estimates;
- prioritizing the software and files for improvement;
- coordinating SIP efforts with other ongoing efforts; and
- assigning improvement tasks and responsibilities to teams.

The SIP team is the focal point for software improvement task assignments, and is responsible for accomplishing the tasks assigned in a timely, efficient manner, according to the plans developed by the task force. Team and task assignments will vary and include such tasks as inventorying software and files, preparing more detailed subplans or microplans, preparing test data sets, and performing actual improvements.

The organizational structure and size of the task force and teams may be different for each improvement increment, release, phase, and task; however, their organization and size should be consistent with the staffing and resource requirements of the activities being performed. Their size will vary depending on the task force or team's functions, the number and type of tasks performed, the size of the tasks, and the skill level of the personnel. The initial task force and teams should be as small as possible, consistent with the size and scope of effort of the SIP, to avoid an unmanageable

number of lines of communication and interfaces. Additional task force and team members may be added after the SIP organization structure is well established, and team functions and assignments firmly decided.

Although the SIP teams can be temporary entities, lasting only as long as the improvement effort itself, it cannot be stressed enough that a SIP is not a part-time job. The task force and teams should be as independent and autonomous as possible and staffed with personnel "dedicated" to the SIP, with no other goals to work for or jobs to work on. The personnel assigned to the task force and teams should represent the full spectrum of the ADP organization including managers, users, programmers, analysts, operators, and auditors.

A SIP task force is comprised of several key persons -- the Project Manager, Coordinator(s), Team Leader(s), Specialty Staff Manager(s), and Consultant(s) [9]. The primary skills required of these members are planning ability, conflict resolution, project management, and software improvement experience. These members should be assigned for the duration of the SIP to preserve continuity of leadership for the SIP. SIP teams consist of Team Leaders and members, whose required skills include familiarity with the ADP environment and applications, technical, clerical, and analytical abilities (i.e., problem resolution versus program logic solving), and some software improvement experience.

The commitment and support of other personnel, external to the SIP task force and team organizations, is also required during the SIP. They include-

- managers for budget and resource allocation;
- users for functional requirements and acceptance testing;
- operators for machine time and executing improvement runs;
- programmers and analysts for inventories and analysis; and
- auditors for test data sets and testing.

a. Project Manager

The Project Manager is the key to the improvement effort, and must be given "exclusive" responsibility and authority to complete the SIP successfully. As a minimum, the Project Manager reports to top management and is charged with-

- planning the project;
- tracking, monitoring, and controlling the project status;
- resolving conflicts and discrepancies;

- establishing, enforcing, and controlling changes, standards, policies, procedures, and guidelines;
- estimating, scheduling, obtaining, and allocating resources;
- supervising subordinates;
- assigning tasks;
- maintaining team skills, morale, and training;
- coordinating the improvement effort with other efforts such as ADPE procurement and installation, daily operations, and software maintenance; and
- acting as a liaison between the SIP and external parties, upper management, and consultants.

b. Coordinator

The Coordinator acts as the single interface between the user(s) and the SIP task force and team(s), reporting to the Project Manager. There may be one Coordinator for all users or each user may have its own Coordinator, depending upon the size and scope of the SIP and the user's involvement. If multiple coordinators are necessary, it is preferable to have a single Coordinator Manager on the SIP task force, thus avoiding conflicts of user interests and too many communication lines. The Coordinator Manager is the single focal point for all the Coordinators and their respective users, and serves as the liaison between the SIP task force, team members, other coordinators, and users.

The primary responsibilities of the Coordinator, and Coordinator Manager, includes-

- participating in the planning process;
- signing off, or accepting for the user, the improved systems;
- requesting and acquiring user personnel for test data set generation and validation, test case and scenario preparation, test result evaluation, and training; and
- evaluating and approving/disapproving requested user changes during the SIP.

c. Team Leader

The Team Leader reports to the Project Manager and interfaces with other Team Leaders and the Coordinator(s). The number of Team Leaders depends upon the size and scope of the SIP and the organizational structure of the SIP task force. The primary purpose of the Team Leader is to alleviate the need for the Project Manager to be bogged down with day-to-day technical activities and decisions. The Team Leader may perform some improvement tasks, but should not handle the tasks that are so critical as to take precedence over, or eclipse, the supervisory functions and responsibilities.

The Team Leader's major functions and responsibilities include-

- assigning and monitoring tasks;
- coordinating tasks between team members and other teams;
- participating in planning strategies;
- developing and ensuring the quality of software improvement and organization standards, guidelines, procedures, and processes;
- providing "technical direction" to team members;
- scheduling of team member training; and
- supervising team members.

d. Specialty Staff Manager

The Specialty Staff Manager is the single focal point and interface responsible for the various specialty functions and activities required in almost all SIP's. The Specialty Staff may consist of one or more people, performing the various activities required and acting in a single position or performing multiple duties. The primary specialty positions, or functions, required are library control, file/DBMS administration, operations support, technical and system support, ADP resources support, and implementation and test support.

A central Librarian acts as a single control point for all program material and enforces the configuration control mechanisms required to ensure that correct versions of the software are improved. A File/DBMS Administrator consults on, and assists in, any file/DBMS design that is required

and aids in resolving any problems that may arise during the file/DBMS improvement. Operations support schedules the software improvement and test jobs, processes the software and file improvements, performs software compilations, and accepts the software improvement work packages for test and production. Technical and system support installs the target system, provides technical support, aids in resolving any problems that may arise during the improvements, establishes programming standards and procedures, and maintains old and new versions of systems software until the software improvement is complete. It also provides specific tool development, evaluation, test, and selection services to the SIP task force and teams. ADP resources support acquires hardware and software for the target environment and/or the SIP, and supports the overall installation planning. Implementation and test support executes and evaluates all software improvement test runs, interfaces with the operations department and operations support, and provides discrepancy reporting.

The Specialty Staff Manager may perform some or all of these activities, or may require a staff to perform these activities. Regardless of the number of people on the Specialty Staff, the Specialty Staff Manager reports to the Project Manager and interfaces with the Coordinator(s) and Team Leader(s), and is responsible for the same major functions and responsibilities as the Team Leader(s).

e. Consultant(s)

The primary responsibilities of an outside Consultant to the task force are to supplement the task force's or member's skills and ideas, and provide a fresh and innovative perspective on the overall SIP and problem resolution. The Consultant should have no vested interests in the outcome of the SIP and provide unbiased and objective responses to task force's or member's queries. The Consultant reports to the Project Manager and interfaces with the Coordinator(s), Team Leader(s), and Specialty Staff Manager(s).

f. Team Member(s)

Improvement team members consist primarily of senior and junior level analysts and programmers who perform most of the actual software improvement tasks such as preparing work packages, improving the software, and documenting the software. They report directly to the Team Leader and interface with other team members.

Exhibit 11 is a sample SIP Task Force/Team Organization Chart for a typical software improvement effort. It is structured to allow the existence of the SIP Task Force, with SIP teams as subordinate organizational or functional entities, and the performance of the improvements by either inhouse or contractor personnel. It is meant to be general in nature and should be amended to fit the organization's specific needs and desires.

4.3 Commitment to a SIP

A commitment to a SIP begins with top management, progresses through the ADP organization, and, ultimately, ends with the user. Top management commitment to the SIP is a major factor for its success, and manifests itself in three forms:

- First, top management must acknowledge that a software problem really exists, and resolve to correct it.

- Second, top management must be willing to "put their money where their mouth is." That is, they must not offer only "lip service," but be willing to devote the resources necessary to implement the SIP. Resources include people, dollars, time, ADPE, tools, and other miscellaneous supplies and materials.

- Third, top management must actively support the SIP and ADP organization by helping to advertise the SIP goals, objectives, benefits, and achievements, and by gaining user involvement and support.

Ideally, top management's decision to undertake a SIP should be supported by a SIP Feasibility Study. The purpose of a SIP Feasibility Study is to provide a cost, benefit, and risk analysis and assessment of the organization's software status, software needs, and different methods for satisfying those needs. Federal agencies should carefully explore alternative methods -- in addition to the traditional alternatives of software redesign or new development -- of satisfying software needs [11], both to be good managers and to choose the most cost-effective, risk-free alternative.

A SIP Feasibility Study provides management with information needed to base their decision on whether or not to initiate a SIP and serves as supporting documentation for this decision. Exhibit 12 illustrates a sample outline for a SIP Feasibility Study. This outline is only a suggestion and should be modified to fit each unique ADP environment and/or situation.

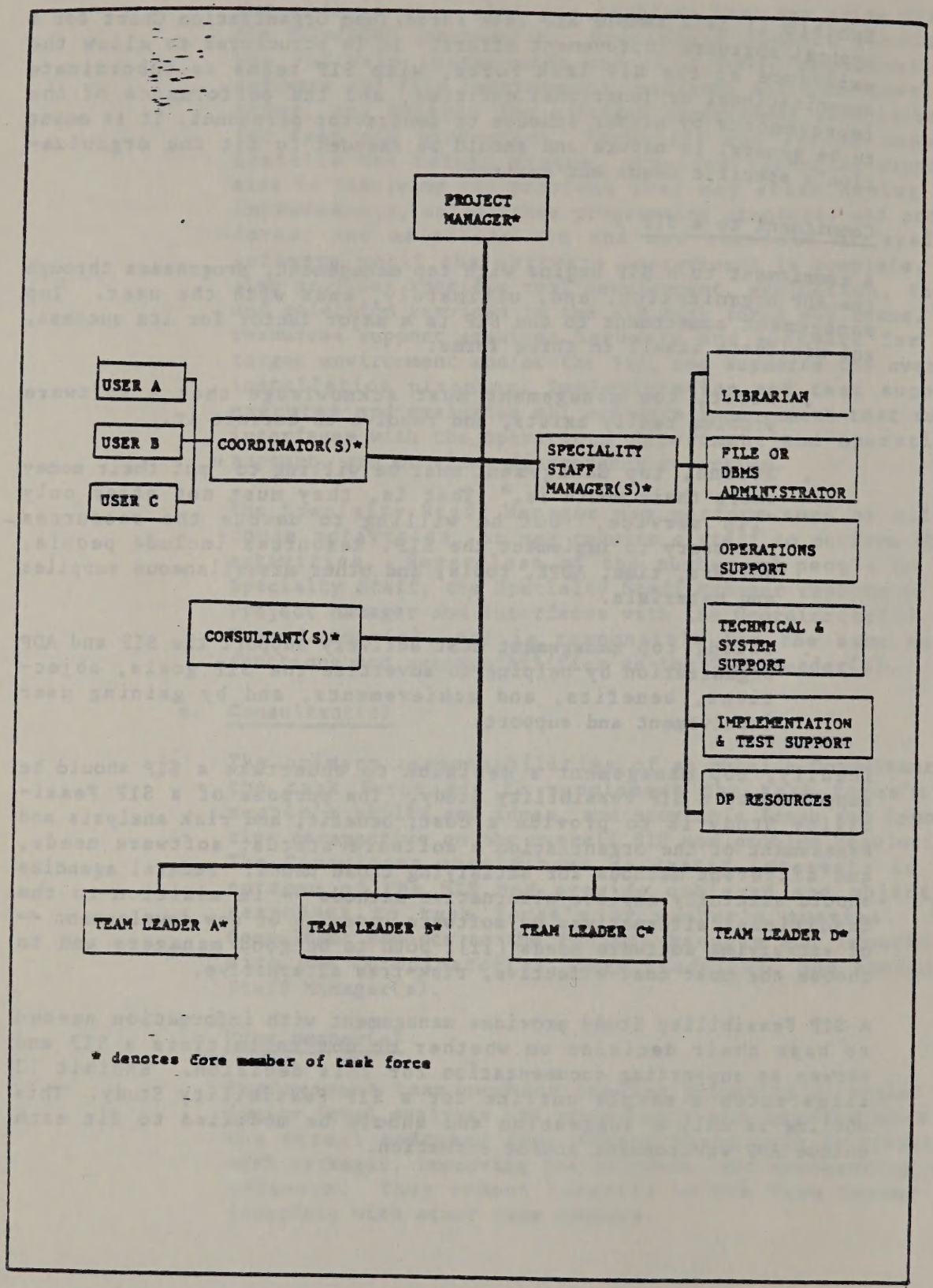


EXHIBIT 11: Sample SIP Task Force/Team Organizational Chart

1. INTRODUCTION
   - 1.1 Purpose and Content of Study
   - 1.2 Scope of Work
   - 1.3 Objectives
   - 1.4 Assumptions and Constraints
   - 1.5 Methodology Used
   - 1.6 Points of Contact

2. DESCRIPTION OF ADP ENVIRONMENT
   - 2.1 Description of Current ADP Environment
     - 2.1.1 Hardware and Facilities
     - 2.1.2 Operating System
     - 2.1.3 Software
     - 2.1.4 Organization, Mission, and Users
     - 2.1.5 Privacy and Security
     - 2.1.6 Processing Modes
   - 2.2 Description of Proposed ADP Environment
     (Same components as 2.1)

3. ANALYSIS OF SOFTWARE PROBLEMS AND SOFTWARE NEEDS
   - 3.1 Description of Software Problems
   - 3.2 Impact of Software Problems
   - 3.3 Description of Software Needs
   - 3.4 Impact of Software Needs

4. ANALYSIS OF ALTERNATIVES TO SATISFY SOFTWARE NEEDS
   - 4.1 Complete Redesign
     - 4.1.1 Description of Alternative
     - 4.1.2 Proposed Approach to Implement Alternative
     - 4.1.3 Analysis
       - 4.1.3.1 Cost Analysis
       - 4.1.3.2 Benefit Analysis
       - 4.1.3.3 Risk Analysis
       - 4.1.3.4 Impact Analysis
       - 4.1.3.5 Summary of Analysis
   - 4.2 Leave System Alone
     (Same components as 4.1)
   - 4.3 Software Improvement Program
     (Same components as 4.1)
   - 4.4 Other
     (Same componenets as 4.1)

5. SUMMARY OF RECOMMENDATIONS
   - 5.1 Recommended Alternative
   - 5.2 Rationale and Justification
   - 5.3 Recommended Implementation Plan/Approach

EXHIBIT 12: Sample Software Improvement Feasibility Study Outline

After a commitment to a SIP has been made, several important activities are required:

- First, analyze the current environment, identifying problems to be resolved and describing the status quo (i.e., "where the ADP organization is today"). Knowing "where you are" is just as important as knowing "where you want to be."

- Second, identify problems to be resolved, improvements in the status quo, and future requirements (i.e., "where the ADP organization wants to be tommorrow"). Do not confuse a "wish list" with a "requirements list." A wish list contains all desirable items and actions and is unconstrained, listing all manner of futuristic items and actions without regard for actual or proven needs. However, a requirements list contains only those items and actions actually deemed (i.e., proven and justified) to be necessary, and will have definite constraints.

- Third, identify those factors that cause a definite constraint on the requirements (e.g., available personnel, budget, skill level of personnel, ADPE, technology, and time). Analyze the impact of these constraints against the requirements list, determining those requirements that can actually be filled.

- Fourth, establish a SIP, including resource allocation and an organizational structure for the SIP's implementation, control, and management.

- Fifth, develop a macroplan for the overall SIP, microplans for the pilot project and each increment, and software improvement release specifications for each release. Describe how the SIP will be implemented, including software improvement strategies, timing considerations and schedules, and detailed problem and solution descriptions.

- Finally, through the organizations established for the SIP, control and manage the evolving software and the software engineering environment.

4.4 Recommendations on Planning and Implementing a SIP

To summarize, there are six key points of a SIP to remember:

- Evolutionary Growth: That is, build on your past software investment as much as possible by purging some of the software, leaving some of it alone, replacing some software with packages, improving most of the software, and redesigning or newly developing only that which is absolutely necessary.

- Incremental Improvement: Minimize risk of failure and make the SIP more manageable by grouping the software, through functional decomposition, into logical subpieces with minimal interfaces.

- Top-Down Planning with Bottom-Up Input: Plan in a hierarchical manner, from a general, overall SIP level (i.e., macroplan), progressing to more specific, increment levels (i.e., microplans). Allow for continual feedback, analysis, and evaluation of improvement plans, results, and methodologies.

- SIP Pilot Project: Prototype the improvements, engineering elements, and methodologies on a small scale to empirically demonstrate the feasibility and success of the improvements, methodologies, and plans; and to help solve, early in the SIP life cycle, any technical problems that may occur.

- Release Specifications: Subdivide the improvements required for each increment into logically grouped improvement activities that can be performed at one time (e.g., conversion, refinement, and enhancement). Develop release specifications to be included in the appendix of each microplan, and which direct and control the improvements to be performed for each release of an increment. Include in the release specifications the specific improvements required for each individual increment component (i.e., system, subsystem, program, module, file, and job stream), the deliverables, standards of performance, and acceptance criteria.

- Engineering Elements (SET): Establish within the SIP, or as a baseline SET, formalized engineering elements (i.e., standards and guidelines, procedures, tools, QA, and training) to be implemented, employed, and enforced by the SIP.

To encompass the six key points of a SIP, the following step-by-step approach to planning and implementing a SIP is recommended. These steps follow the six aforementioned key commitment activities required:

- Develop a macroplan for the overall SIP.

- Select a pilot project as an increment, to prototype the SIP.

- Establish the engineering elements, preferably as a baseline SET, for use by the SIP.

- Develop a microplan for the pilot project.

- Prepare software improvement release specifications for each improvement release of the pilot project.

- Prototype the SIP for the first improvement release of the pilot project.

- Measure and evaluate the SIP pilot project results and methodologies.

- Revise and update the macroplan and SIP pilot project microplan, and the engineering elements or baseline SET.

- Expand the SIP to the remaining software, incrementally phasing into an organization-wide SIP.

- Divide the remaining software into increments (i.e., logically grouped subpieces with minimal interfaces).

- Divide the increments into releases (i.e., logically grouped improvement activities that can be performed at one time).

- Develop microplans for subsequent increments, and develop software improvement release specifications for each increment's releases.

- Accomplish first, the improvements for the most critical increments, or for the increments with the greatest payoff or return on investment; then, accomplish the improvements of the remaining increments.

- Continually update and revise the macroplan, and each increment's microplan, until an organization-wide SIP is well-established and fully implemented.

4.5 Summary

While the SIP guidelines presented herein may not seem to be "earth-shaking," they are a less risky, more formalized means of modernizing the organization's information processing and have been found to be "tried-and-true." The use of these SIP guidelines is strongly encouraged and, in concert with a strong SET, will promote more uniform, thorough, cost-effective, and efficient software and software engineering activities.

The most successful organizations have recognized that their software is a key asset, which must be developed, managed, controlled, and maintained with as much care and attention as their other important assets. That is why these organizations have invested, or are investing, significant resources into SIP's, using principles similar to those described here. The experiences of these organizations should not be considered unique. The concept of a SIP is indeed a valid alternative to the two traditional choices of

"don't-touch-it-or-it-will-fall-apart" and "tear-it-down-and-start-anew." Unless the functions are changing, substantial redesigns may not be necessary, and a SIP may solve immediate, as well as long-range, ADP problems. It is an alternative with principles and procedures applicable to most Government agencies and must be given serious consideration.

Several organizations have successfully established and undertaken SIP's. Some examples of these organizations are the Office of Personnel Management (OPM) in Washington, DC; Raytheon Service Company in Boston, MA; NCR Corporation in San Diego County, CA; and the San Diego County Department of Education in San Diego, CA.

Several years ago, OPM undertook a SIP with gratifying results. Many of its ADP systems were developed in Assembler language for an RCA Spectra 70/45 and, when converted to COBOL, still reflected the second generation logic of the earlier Assembler code. OPM decided to adopt a controlled system improvement approach, migrating in steps from a second to third generation system. The decision was also made to convert the Assembler code to ANSI COBOL to simplify maintenance and enhance portability considerations [12].

Similarly, in the late seventies and early eighties, Raytheon undertook a SIP with the primary objective of developing, implementing, and perfecting a reuseable code methodology for accelerating applications development. By using reuseable code, 40 to 60 percent of the redundancy in their business applications development was eliminated, and maintenance was substantially improved [13].

In late 1976, the NCR Corporation undertook a large scale Quality Improvement Program (QIP) for a major set of systems software for over 103 separate products. This software set included operating systems, compilers, peripheral software, data utilities, and telecommunications handlers, and totaled over 1.3 million lines of source code. The QIP was initiated to provide improvements in the software base and to take advantage of recent advances in the state-of-the-art of software engineering. NCR found several major favorable effects resulting from the QIP, such as a substantial reduction in outstanding problems in the software base, a reduction in the average number of error reports per month, total elimination of problem backlogs, and a significant reduction in late responses to problem reports. All of these improvements are reflected in an improved perception of the quality of the software and allowed NCR to make a very substantial redirection in funds from support of existing products to the development of new ones [4].

Finally, the San Diego County Department of Education's ADP data center cut its maintenance time by 70 percent as a result of a SIP. This startling reduction in the data center's program-maintenance load has resulted primarily from the decision to adopt and convert to structured design and programming techniques, with ongoing and formalized ADP training in these same areas. While the primary

emphasis in this effort was on redesigning and replacing the existing systems, rather than salvaging the existing systems, the six key principles of a SIP were basically adhered to [14].

Besides the organizations that have established and undertaken successful SIP's, several Federal organizations are currently in the initial steps of establishing SIP's. Several of these organizations are the Social Security Administration (SSA) in Baltimore, MD; Veterans Administration's (VA) Data Processing Center (DPC) in Austin, TX; and Defense Mapping Agency (DMA) in Washington, DC.

The SIP being established by the SSA is a prime example where the two traditional alternatives are infeasible. SSA is in the process of transitioning its more than 16,689 computer programs, containing over 11 million lines of code, from a "survival" mode to a state-of-the-art environment. To leave the systems alone will only result in further ADP system deterioration and seriously jeopardize the Agency's ability to perform its basic mission. Conversely, to redesign all software would be extremely risky, require maximum reinvestment of resources, and require more time than SSA has to survive the current crisis. Therefore, the key is an incremental, evolutionary improvement approach, aimed at a recovery of SSA's heavy investment in its software, and the ability to take advantage of new ADP technological advances [15].

The VA's Austin DPC is another example where top management has recognized that the efficiency and effectiveness of their mission is dependent upon their software. The Austin DPC has over three million lines of code of application software. Like software in most Government agencies and industry, this software was not developed overnight; rather, it has evolved over many years, with layer upon layer of modification making it even more complex, unmangeable, and unmaintainable. Together with a hardware upgrade and SET initiative, the Austin DPC is initiating a SIP to cut escalating ADP costs, improve the quality of service to veterans, and better support far-reaching management decisions [16].

The DMA is currently in the initial stages of a 5-year program to upgrade its software and modernize it software production practices. The ultimate objectives of this SIP are to increase productivity, improve software quality, and standardize software development practices. DMA's SIP encompasses three major areas: introduction of a modern programming environment, improvement of existing software, and upgrading developmental and managerial skills to support the new environment [17].

Several additional organizations establishing SIP's are Tupperware in Orlando, FL; Ford Aerospace and Communications Corporation in Sunnyvale, CA; and the New Jersey State Government.

Thus, from the preceeding discussion, it should be clear that establishment of a SIP is a must for organizations who can no longer afford outdated and inefficient information processing, want to combat the software crisis, want to stop "software "senility" in its tracks, and, on the whole, need to modernize their software and software engineering technologies.

REFERENCES

1. Software Improvement - a Needed Process in the Federal Government, Office of Software Development, General Services Administration, Report No. OSD-81-102, June 3, 1981.

2. Establishing a Software Engineering Technology, Federal Software Testing Center, General Services Administration, Report No. OSD/FSTC-83/014, June 1983.

3. Long Range Plan, Office of Software Development, General Services Administration, Report No. OSD-82-104, April 9, 1982.

4. Woodmancy, Donald A., "A Software Quality Improvement Program," NCR Corporation, San Diego, CA, IEEE Catalog No. 79CH1479-5/C, 1979.

5. "Designing Software with Maintenance in Mind," Datapro Report No. AS75-075-101, DATAPRO Research Corp., Delran, NJ, March 1982.

6. Jensen, Randall W. and Tonies, Charles C., Software Engineering, Prentice-Hall Inc., Englewood Cliffs, NJ, 1979.

7. Stucki, Leon; Brown, John; and Hammond, Linda; "DMA Modern Programming Environment Study," Report No. RADC-TR-79-343, Rome Air Development Center, Griffiss AFB, NY, January 1980.

8. Federal Conversion Support Center Conversion Cost Model (Version 2), Federal Conversion Support Center, General Services Administration, Report No. GSA/FCSC-82/001, June 1, 1982.

9. Managing Application Conversion Projects, Workbook, IBM Independent Study Program, IBM.

10. Conversion Management Guidelines, Initial Draft Final Report, National Bureau of Standards, August 31, 1981.

11. Federal Agencies Could Save Time and Money With Better Computer Software Alternatives, General Accounting Office, Report No. GAO/AFMD-83-29, May 20, 1983.

12. Cooper, Roger, "Upgrading Federal Computers Through Existing Systems," Government Executive, August 1979.

13. Lanergan, Robert G. and Poynton, Brian A., "Reusable Code: The Application Development Technique of the Future," Raytheon Service Company.

14. Beeler, Jeffry, "Manager Cuts Maintenance Time by 70%," Computerworld, January 31, 1983.

15. Systems Modernization Plan, Social Security Administration, U.S. Department of Health and Human Services, February 1982.

16. Software Improvement Program (SIP) Macroplan - Draft, Veteran's Administration, Austin, TX Data Processing Center, May 1983.

17. Stroup, Opal R., DMA Software Improvement Program, Talking Paper for Federal DP Expo (Session E-2) "Dealing with Obsolescence: Conversion and Upgrading," DMA, U.S. Naval Observatory, Washington DC, April 12, 1983.

GLOSSARY OF TERMS

ACCURACY: Degree to which software produces outputs that are sufficiently precise to satisfy their intended use.

ADP: Automatic Data Processing.

ADPE: Automatic Data Processing Equipment.

CONCISENESS: Minimization of the amount of code necessary to perform the desired function.

CONVERSION: Transformation of software, without functional change, to standardize it and make it environment independent, or to permit its usage on a replacement or changed ADPE system or teleprocessing service.

DBMS: Data Base Management System.

DEPENDABILITY: Extent to which software can be relied upon to consistently function in a specified manner at specific times.

DPC: Data Processing Center.

DMA: Defense Mapping Agency.

EFFICIENCY: Economical performance of functions and fulfillment of purpose, without waste of resources (e.g., funds, time, personnel, computer time, main memory, communication channel capacity, or materials).

ENHANCEMENT: Optimization of the value, quality, efficiency, and effectiveness of the software to enable easier technical redesigns and addition of modern "technological" features and capabilities, and more efficient and effective use of resources.

FCSC: Federal Conversion Support Center (part of Office of Software Development, Office of Information Resource Management, General Services Administration).

FSEC: Federal Software Exchange Center (part of Office of Software Development, Office of Information Resource Management, General Services Administration).

FSTC: Federal Software Testing Center (part of Office of Software Development, Office of Information Resource Management, General Services Administration).

FLEXIBILITY: Ease with which software can be changed or revised.

GENERALITY: Extent to which software can be used for a variety of changing functions without introducing revisions.

GSA: General Services Administration.

IMPLEMENTABILITY: Extent to which, and ease with which, the software is able to be put into production or operation.

INCREMENT: Logically grouped subpieces of software, by functional decomposition, with minimal group interfaces.

INDEPENDENCE: Degree to which code is free of architecture, device, or environment dependencies, or vendor compiler extensions.

MACROPLAN (for SIP): A hierarchy of plans and subplans (e.g., microplans, personnel plans, and procurement plans) that address, in a general way, the need for, requirements of, and strategies for the overall organization-wide SIP.

MAINTAINABILITY: Degree to which code facilitates updating to satisfy new requirements, rectify deficiencies, or correct errors. Implies that the code is, to some degree, understandable, testable, modifiable, modular, concise, and simplistic.

MICROPLAN (for SIP): A hierarchy of subplans that addresses each SIP increment; provides step-by-step instructions on what is to be accomplished in each increment, and how it is to be accomplished; designates specific task assignments and responsibilities; and specifies definitive schedules and milestones. Appendices of Microplan contain detailed software improvement release specifications for each of the increment's releases.

MODIFIABILITY: Ability to be changed or revised for various reasons or purposes with relative ease. Implies code is either flexible or general.

MODULARITY: Extent to which software is built in small, manageable, single-function pieces of code that are independent and self-contained. Implies that code is structured in such a way that there is only one entry and exit point to each module, variables are "visible" only in the module in which they are used, and the inputs/outputs and software functions are isolated.

OPM: Office of Personnel Management (formerly Civil Service Commission).

OSD: Office of Software Development (part of Office of Information Resource Management, General Services Administration).

PILOT or PROTOTYPE PROJECT (for SIP): A sample and experimental SIP project that empirically demonstrates the feasibility and success of the improvements and the methodologies, and helps solve, early in the SIP life cycle, any technical problems that may occur.

PORTABILITY: Extent to which software can be moved to, and operated easily on, other computer configurations and operating environments. Implies the existence, to some degree, of uniformity, independence, and reuseability.

PROCEDURES: Standard methods of doing business within an organization (i.e., standard operating procedures).

QIP: Quality Improvement Program.

QUALITY ASSURANCE (QA): Formal process of measuring or evaluating the degree to which software, or a software process or activity, meets standards and/or prescribed requirements.

QUALITY: Measure of software's excellence, worth, or value against some ideal or standard. Properties by which quality can be defined or measured are useability, reliability, dependability, accuracy, implementability, efficiency, portability, uniformity, independence, reuseability, maintainability, testability, modifiability, modularity, conciseness, and simplicity.

REDESIGN/NEW DEVELOPMENT: Any change to software that involves a change in the functional specifications of that software. There is a fine line of distinction between software redesign and new development, depending on the extent of the redesign and the amount of logic, analysis, and functionality that can be salvaged from the existing software. The greater the software redesign undertaking, and the more the functionality of the software changes, the more synonymous redesign and new development become.

REFINEMENT: Modernization of the software to a state-of-the-art status and improvement of software maintainability and programmer productivity.

RELEASE: Logically grouped improvement activities that can be performed at one time (e.g., conversion, refinement, and enhancement).

RELEASE SPECIFICATIONS: Document that controls and directs the improvements to be made to an increment during each release. It describes the specific problems to be resolved, and improvements to be made to each of the increment's components, on an individual system, subsystem, program, module, jobstream, and file basis.

RELIABILITY: Extent to which software correctly and satisfactorily performs its intended functions. Implies the software is dependable, accurate, and implementable.

REUSEABILITY: Ability of software to be used over and over again for various situations or reasons (e.g., common routines used as building blocks for software development).

SET: Software Engineering Technology.

SIMPLICITY: Measure of the degree of complex decision making present in a piece of code.

SIP: Software Improvement Program.

SOFTWARE AND TECHNOLOGY MODERNIZATION PROGRAM (STMP): A "double-barreled" approach to resolving the software problems by establishing both a SIP and a SET.

SOFTWARE ENGINEERING TECHNOLOGY (SET): A synchronized group of five engineering elements (i.e., standards and guidelines, procedures, tools, quality assurance, and training), that direct and control all software activities throughout the software's life cycle, and for different software engineering or re-engineering purposes.

SOFTWARE IMPROVEMENT: The application of today's methodologies to yesterday's software to meet tommorrow's needs. Retrofit actions (i.e., taken after the fact), ranging from simple translation of code to complete re-engineering of existing systems, taken to modernize its value, quality, effectiveness, and efficiency. Software improvements preserve the value of past software investments as much as possible, create an improved foundation upon which to maintain older systems and build new systems, and enable the organization to capitalize on modern technological advances in the field.

SOFTWARE IMPROVEMENT PROGRAM (SIP): An incremental and evolutionary approach to the modernization of existing software to maximize its value, quality, effectiveness, and efficiency. It preserves the value of past software investments and, at the same time, increases the reliability, efficiency, portability, and maintainability of the software to create an improved foundation upon which to maintain older systems and build new systems.

SSA: Social Security Administration.

STAGES, SOFTWARE LIFE CYCLE: The discrete phases through which software passes during its existence -- Requirements Definition and Analysis, Design, Programming, Validation, Operation, and Review. Though not exactly the same as the phases/stages identified in FIPS Publication 38, these six stages are similar, if not synonymous, in nature and function.

STANDARDS: Those which are established by authority, custom, or general consent as a basis for comparison.

STMP: Software and Technology Modernization Program.

TASK FORCE/TEAMS: Organizational structure to establish, plan, implement, manage, control, and accomplish the SIP. The task force is the focal point of all SIP planning and management, and consists of several key persons -- the Project Manager, Coordinator(s), Team Leader(s), Specialty Staff Manager(s), and Consultant(s). The team is the focal point for the accomplishment of all SIP task assignments, and consists of Team Leaders and members.

TESTABILITY: Ease with which software changes can be demonstrated, in a quantitative manner, to be correct.

TOOLS: Software packages, or computer programs, that aid in the specification, construction, testing, analyses, management, documentation, and maintenance of other computer programs.

TRAINING: A process or method to lead or direct growth, or form by instruction, discipline, or drill, thus creating, through some type of learning experience, a permanent change in a person's behavior, so the individual reliably performs in a certain prescribed manner.

UNDERSTANDABILITY: Extent to which the purpose and function of the code, and its documentation, are easily discerned by the reader.

UNIFORMITY: Existence of standardization such as naming conventions, code alignment, and common formats and interfaces.

USEABILITY: Extent to which software has worth, and is reliable, efficient, portable, and maintainable.

VA: Veteran's Administration.

GSA Office of
Software Development

5203 Leesburg Pike, Suite 1100
Falls Church VA 22041

Official Business
Penalty For Private Use $300

Postage and Fees Paid
U.S. General Services Administration
GSA-361

THIRD CLASS

APPENDIX B

BLM SOFTWARE CONVERSION AND ASSESSMENT SURVEY FORM

Software Conversion and Assessment Survey
for the Bureau of Land Management

1 Application profile and general information:

This section provides an overview of the application. It describes the role of this application with respect to BLM's mission, existing plans concerning this application, and the relationship of this application to other BLM applications and outside users or systems.

1.1 Application identification:

Application Name _____ Acronym _____ Chargeback Code[1] _____

1.2 Point of contact for information on these forms:

Please identify the person to call if there is a question concerning information provided for this application. Also, please circle the name of the person completing the form.

Main Contact: _____ BLM Office: _____ FTS Phone: _____ Comm Phone _____

Backup Contact: _____ BLM Office: _____ FTS Phone: _____ Comm Phone _____

1.3 BLM program(s) supported by this application:

_____

1.4 Application purpose and key products:

Briefly describe the application's purpose in terms of it's key products. For example, the Range Management Automated System (RMAS) uses planned use, fee collection, and descriptive information on grazing land to produce grazing statistics, bills, and management reports.

_____

_____

_____

Notes:

1) Chargeback code refers to the code used by the DSC Chargeback Application to charge ADP resource usage (e.g. CPU time, CPU memory, and disk) back to specific applications. For example, MV is the Chargeback Code for the Motor Vehicles application.

1.5 If this application were to be replaced, would it be rewritten, superceded, or discontinued given its current condition and the applications currently planned? When would this most likely happen? (Answer A and B. Answer C if applicable.):

By superceded, we mean this application would be absorbed by another application. If it is rewritten, the application retains a separate identity. This information will be used to assess if it is cost effective for BLM to invest in major enhancements to this application.

A ☐ Rewritten ☐ Superceded ☐ Discontinued

B ☐ Within 3 Years ☐ Within 5 Years ☐ Within 10 Years ☐ None Planned

C If superceded, by which application:

☐ ALMRS ☐ GIS ☐ DOI-wide system ☐ Other _____

1.6 Summarize how the Bureau's operations would be affected if this application were not successfully converted (e.g. rangeland permits would be calculated and issued by hand.):

_____

_____

_____

_____

1.7 Summary of major planned enhancements, modifications, or maintenance

By major, we mean changes which will affect the support requirements of the application in terms of technical capabilities, such as changing from sequential files to a DBMS, from line-by-line data entry to screen data entry, or from batch to online processing mode.

1.7.1 Present to 1988:

_____

_____

1.7.2 1988 to 1995:

_____

_____

1.8 This application runs (Answer both A and B):

A ☐ In Production ☐ On User Request ☐ Executed By User Directly

B ☐ Online ☐ Batch ☐ Both Batch and Online Components

1.9 Interfaces with other BLM applications and outside systems:

☐ No

☐ Yes, as listed below:

Application or outside system	Input or Output [1]	Frequency (e.g. daily, weekly, monthly)	Method of Interface (e.g. tape, online, cards)	Volume (Please label units)	For outside systems, list agency/organization, hardware, and location

Notes:

1) Enter INPUT or OUTPUT: Input means this application receives data from another application or outside system. Output means it is providing data to them.

2 Documentation and testing summary:

This section addresses the quality of current documentation and the status of current testing. The quality of documentation and the amount of test data already available has a significant impact on the amount of effort required for conversion

2.1 Documentation

Type	Custodian/Phone	Last Update	Value (0-10) [1]	# of Pages
Functional Requirements				
System/Subsystem Spec				
Database Spec				
Program Spec				
Program Documentation				
Operations Documentation				
User Documentation				
Other (Specify) _____				

2.2 Testing

2.2.1 Does test data exist for this application?

☐ No ☐ Yes

If yes, date of last update _____ What percent of the program logic is tested by this data? _____

2.2.2 Are there separate test and production versions of the application code?

☐ No ☐ Yes

Notes:

1) VALUE: The software conversion model requires an estimate of documentation value according to the following scale:

10 - Complete and up-to-date
9 - Complete but somewhat out-of-date
8 - Extensive amount which is incomplete but useable and up-to-date
7 - Extensive amount which is incomplete and out-of-date but useable
6 - Extensive amount which is incomplete, not useable, and out-of-date
5 - Moderate amount exists which is incomplete but useable and up-to-date
4 - Moderate amount which is incomplete and out-of-date but useable
3 - Moderate amount but incomplete, not useable, and out-of-date
2 - Very little exists but is up-to-date
1 - Very little exists and is out-of-date
0 - No documentation

3 Input, Output, and Data Storage Summary

This section collects information on data input, output, and storage which will effect the size and complexity of the conversion effort. Growth information is included since not all the applications will be converted at once and will continue to grow prior to conversion.

3.1 Input summary:

Input can be in the form of records, transactions, or both. Transactions are online interactions between a data source (e.g. CRT or another application) and an application. Online applications may have both transactions and records; batch applications have only records.

3.1.1 Summary information:

SOURCE[1]						MONTHLY							GROWTH[7]		
BLM Site	Other Applic	Type of Process[2]	Device Used[3]	Freq[4]	Record Length (in char)	# of Records Aver	# of Records Peak	# of Setups Aver	# of Setups Peak	# of Trans Type	# of Trans Aver[6]	# of Trans Peak	Annual Growth %	Growth Surge %	Growth Surge Year

Notes:

1) Sources: BLM site or application providing input. Be specific. If NMSO input is keyed by DSC, enter NMSO-DSC
2) Process Type: sample entries would be ONLINE, BATCH, KEYED (CARDS, TAPE OR DISK))
3) Device Used: sample entries would be CRT, OCR, DIGITIZER, KEY-TO-DISK, KEY-TO-TAPE, or RJE
4) Frequency: sample entries would be DAILY, WEEKLY, MONTHLY, OR QUARTERLY
5) Setups: the number of physical tape or disk mounts required by the application
6) Type: L for line-by-line online transaction, S for screen transaction, and blank for batch applications
7) Growth: anticipated growth as a percent of current input size. If an unusual increase (a 'surge' or 'spike') is anticipated due to special circumstances (e.g. a new function becomes available), enter the year and size of the anticipated surge.

3.1.2 Method of correcting input (e.g. operator intervention, error reports, rerun application):

_____

_____

3.1.3 Comments/problem areas concerning input:

_____

_____

3.2 Output summary

3.2.1 Output summary information

Delivery Point [1]	Output Device [2]	Freq [3]	MONTHLY Lines, Pages, or Frames [4] Aver	Peak	MONTHLY Pages of Special Forms Aver	Peak	GROWTH [5] Annual %	Growth Surge %	Year

3.2.2 Comments/problem areas concerning output:

Notes:

1) Delivery point: BLM Location or application receiving output. Be specific. If NMSO output is printed by DSC and sent to NMSO, enter NMSO-DSC
2) Output Device: Give device and its location. Sample entries would be PRINT-DSC, PLOT-MSO, CRT-WSO, FICHE-DSC, TAPE-DSC
3) Frequency: Sample entries would be DAILY, WEEKLY, MONTHLY, OR QUARTERLY
4) Use L for lines, P for pages, and F for frames (e.g. 50P for fifty pages per month).
5) Growth: anticipated growth as a percent of current output size. If a growth surge is anticipated, enter the year and size of the anticipated surge.

3.3 Data storage summary

3.3.1 Flat file storage information:

Device:	# of Files	Llinks	# of Fixed Record Length	# of Variable Record Length	GROWTH Annual %	GROWTH Growth Surge % Year	
Disk Pack							
Cartridge Disk							
1600/6250 BPI 9-track Tape Drive							
800/1600 BPI 9-track Tape Drive							
800/1600 BPI 7-track Tape Drive							
Other (Specify)							

3.3.2 Database storage information:

DBMS:	# of Databases	# of Subschemas	# of Rec Types	Llinks Used	GROWTH Annual %	GROWTH Growth Surge % Year	
DM-4							
INFO-6							
ASPEN/2							
Other							

3.3.3 Data security requirements:

☐ Data Encryption

Comprimise of this data, such as oil field production and royalty information, could result in very serious financial or legal consequences for BLM such as oil field production and royalty information.

☐ Sensitive Data

Data is private and confidential, such as personnel, payroll, or BLM internal planning information; or could result in some financial loss or legal risk to BLM but on a lesser scale than data requiring encryption.

☐ Controlled Data Access

Data should be viewed and changed only by a select group but poses little risk if compromised.

☐ No Security Needs

Data can be accessed and changed by anyone with access to the computer.

☐ Other (Please Explain)

_____

_____

4 Application code and Job Control Language (JCL) summary:

This section collects information on the languages and the amount of JCL used by the application. The languages used and the extent to which they are used effects the complexity of the conversion effort.

4.1 Application code:

Language	Total # of Programs	# of Online	# of Batch	Total Lines of Code	GROWTH Annual %	GROWTH Growth Surge %	GROWTH Growth Surge Year
COBOL							
FORTRAN							
BASIC							
DM-4							
ASPEN/2							
INFO-6							
SCREENWRITER							
Assembler							
Other (Specify)							

4.2 JCL summary:

# of Job Streams	JCL Library Name(s)	Lines of JCL	GROWTH Annual %	GROWTH Growth Surge %	GROWTH Growth Surge Year

5 System Utilities and Software and Special Equipment

This section collects information on the systems utilities or software and special equipment used by this application to provide a perspective on the relative importance of these utilities and equipment to BLM's overall architecture, and the impact replacement would have on BLM's ADP applications.

5.1 System Utilities and Software

Product	# of Programs Using It
BSC Transport Facility	
Comm. and File Transfer Facility	
Data Entry Facility	
DM-4 Transaction Processor	
INFO-6	
OAS Document Processing	
OAS Records Processing	
Production Facility (UPF)	
SAS	
SPSS	
SORT/MERGE	
TPS-6 SCREENWRITER	
TPS-6 Transaction Processor	
2780/3780 WKSTN Facility	
Other _____	
_____	
_____	

5.2 Equipment

5.2.1 Approach for keyed input

☐ Line-by-line

☐ Full-screen

☐ Either, depending on terminal make and model. Terminals for which this application supports full-screen are:

Make	Model
_____	_____
_____	_____
_____	_____
_____	_____

☐ Either, depending on (please explain) _____

_____

5.2.2 Special equipment required by this application

☐ Optical Character Reader ☐ Plotter ☐ Polycorder ☐ Cassette Reader

☐ Card Reader/Punch ☐ Digitizer ☐ Wide Printer ☐ Microfilm or Fiche

☐ Other _____

_____

_____

5.2.3 Special considerations

Are there any special conditions or characteristics of this application which may complicate or prohibit its conversion to the new technical architecture? _____

_____

_____

APPENDIX C

BLM SOFTWARE APPLICATION ASSESSMENTS

This Appendix contains the application assessment forms for the following BLM Bureau-wide Applications:

- Adopt-A-Horse
- Automated Fleet Management
- Cadastral Survey Field Notes
- Checks to Treasury
- Financial Management Edits
- Financial Management Reports
- Financial Management Year-End
- Inventory Data
- Lease Management
- Material Sales
- Operating Budget
- Program Management
- Property Management (Automated Personal Property)
- Reimbursable Billing
- Summer Hire Program
- USFS Forest Inventory
- Waterpower System
- Wildfire Reporting
- Wildlife Information

The data was collected from BLM ADP support staff through written surveys and phone interviews. For more information on the surveys, the reader is referred to the Software Conversion Study, delivered to BLM by AMS in June, 1986.

<< BLM SOFTWARE APPLICATION ASSESSMENT >>

Part 1: General Information

Application Name: Adopt-A-Horse

Contact: Charles Boyer (user rep), Virginia Crosby (programmer)

Age: Approximately 1 year old

Part 2: Technical Characteristics

Operational Mode: Runs in scheduled production and on user requests. Includes both batch and timeshare components.

Language(s) and Size:

FORTRAN	195 programs	43,000 LOC
ASPEN/2	15 programs	1,500 LOC

Reliability Required: Nominal

Data Base Size: Very high ratio of data to LOC (2,663 ch/LOC)

Complexity: High

Execution Time Requirements: Nominal, less than 50% of CPU time available

Core Storage Requirements: Nominal, less than 50% of core available

Machine Volatility: Low

Turnaround Time: Nominal, generally within 4 hours

User Population/Location: Used primarily at Resource Area Offices with summary information used by District, State, and Washington Offices

BLM Program(s) Supported: Wild Horse and Burrow Program

Method of Data Entry: On-line data entry

Part 3: Supportability

Language Use:

The Adopt-A-Horse System uses two languages: FORTRAN 77 and ASPEN/2. The FORTRAN programs are primarily within ANSI standards with only minor Honeywell specific extensions used. ASPEN/2 is machine specific and non-standard.

If this application were converted, parts of the FORTRAN programs and the ASPEN/2 programs would have to be rewritten for execution in the target environment.

Design and Development Methodology:

The FORTRAN programs conform to BLM's structured coding and design standards and are modular. The average module is two pages. Flowcharts, data flow diagrams, and N-S diagrams are available.

Documentation:

Overall documentation is average to weak (3). Operations and User documentation is strong (9), but other types of documentation is virtually nonexistent.

Data Standards:

The application does not use the Data Element Dictionary or any other formal standards.

Test Data:

No test data exists for this application.

Conversion History:

No previous conversion history.

Part 4: Meeting User Requirements

Weaknesses in Meeting User Needs:

Users have complained that the application is too slow (because of the timesharing involved). Steps are being taken to address this problem at this time (see next comment).

Planned Improvements:

The application is currently being converted to run on a microcomputer. Field Offices may use the application locally and send floppies to DSC for batch input to update a central database.

Development Cost Ceiling:

Development Person-Months: 125
Cost (@ $3,100/month): $387,500

<< BLM SOFTWARE APPLICATION ASSESSMENT >>

Part 1: General Information

Application Name: Automated Fleet Management System

Contact: Marvin Gutwein

Age: Approximately three months (replacing Motor Vehicle System)

Part 2: Technical Characteristics

Operational Mode:	Primarily on-line, some batch
Language(s) and Size:	COBOL (DM-IV embedded) 80 programs 106,000 LOC (these figures are from the MVS application, they should be fairly similar)
Reliability Required:	Very Low
Data Base Size:	Nominal
Coding Complexity:	Nominal
Execution Time Requirements:	Nominal
Core Storage Requirements:	Nominal
Machine Volatility:	Low
Turnaround Time:	Very High
User Population/Location:	State, District, and Resource Area Offices
BLM Program(s) Supported:	Property Management
Method of Data Entry:	keyed on-line

Part 3: Supportability

Language Use:

The COBOL 74 and DM-IV are within ANSI standards.

Design and Development Methodology:

Developed using structured coding techniques. The code conforms to the Computer Applications Division Standards.

Documentation:

Still working on the documentation. Planning on the documentation being very complete.

Data Standards:

Uses the Data Element Dictionary and consistent naming conventions.

Test Data:

Yes. 100% of the program logic is tested by the test data.

Conversion History:

No previous conversion history.

Part 4: Meeting User Requirements

Weaknesses in Meeting User Needs:

In response to the original version, users requested additional reports be developed, additional query options be added, and the process be quicker.

Planned Improvements:

The programmers plan to address all of the above user requests. The process speed is generally limited by the telecommunications, but in some instances, some steps can be taken out (e.g., skip a screen that tells user a process is being done, and just go on to the next screen on the assumption that it is).

Development Cost Ceiling:

Development Person-Months: 292
Cost (@ $3,100/month): $905,200

<< BLM SOFTWARE APPLICATION ASSESSMENT >>

Part 1: General Information

Application Name: Cadastral Survey Field Note System

Contact: John Cheatwood

Age: Pre-1972

Part 2: Technical Characteristics

Operational Mode: Batch operation by user directly

Language(s) and Size: COBOL 15 programs 24,000 LOC
ASPEN/2 1 program 290 LOC

Reliability Required: Nominal

Data Base Size: Very high

Complexity: Very low

Execution Time Requirements: Nominal

Core Storage Requirements: Nominal

Machine Volatility: Low

Turnaround Time: Nominal

User Population/Location: Surveyors in the field offices (RA or DO); SO records staff; and DSC data entry

BLM Program(s) Supported: Cadastral Survey

Method of Data Entry: Forms completed in the field and then indexed and keyed at DSC.

Part 3: Supportability

Language Use:

COBOL 74 is the primary language and the application code conforms to ANSI standards. There is no machine or device dependent COBOL although there is one large ASPEN/2 program and ASPEN/2 is machine dependent.

Design and Development Methodology:

This application was developed before the current application development methodologies were implemented at BLM. It is not modular and does not adhere to any particular coding standards. The code contains logic which branches around entire sections of "dead code" (code which is never executed).

Documentation:

Documentation is weak (3 on a scale of 10). It is strongest in user documentation but has no functional requirements or system specification documentation.

Data Standards:

Data naming conventions are consistent within the application. One of the programs may use the Data Element Dictionary, but most do not.

Test Data:

None reported.

Conversion History:

Straight code conversion from the Burroughs in 1978 to the Honeywell without modification.

Part 4: Meeting User Requirements

Weaknesses in Meeting User Needs:

There is currently no mechanism for the users to enter or update data on-line. All data entry is done by DSC. Also, ISO, USO, and WSO are not on the system yet. The data is available in the field note books but has never been transcribed into the 104 character format required by the application. The data for ISO is currently being transcribed for input into the application.

Planned Improvements:

There are no current plans to improve the system beyond normal maintenance. If this application were to be modernized, the BLM staff supporting it would recommend a complete rewrite.

Development Cost Ceiling:

Development Person-Months: 41
Cost (@ $3,100/month): $127,100

<< BLM SOFTWARE APPLICATION ASSESSMENT >>

Part 1: General Information

Application Name: Checks to Treasury

Contact: Jan Geiger

Age: Less than a year; redesign and rewrite of old system.

Part 2: Technical Characteristics

Operational Mode: Batch and timeshare

Language(s) and Size: COBOL 41 programs 50,000 LOC

Reliability Required: High

Data Base Size: High

Coding Complexity: Very low

Execution Time Requirements: Nominal

Core Storage Requirements: Nominal

Machine Volatility: Low

Turnaround Time: High

User Population/Location: DSC and WO

BLM Program(s) Supported: Finance

Method of Data Entry: Key entry, also file transfer from WO and Treasury

---

Part 3: Supportability

Language Use:

This application is written in COBOL 74 within ANSI standards. Only one program has machine dependent code -- a routine to perform data format conversions for creating data for an IBM system. No vendor-specific languages are used.

Design and Development Methodology:

The design and code were developed using structured analysis and coding techniques. The code conforms to the standards of the Computer Applications Division handbook. Code was developed using Quality Assurance techniques such as code walkthroughs.

Documentation:

Overall the documentation was rated 4 on a 10-point scale. Documentation is still being developed in the user and operations areas but the functional requirements and system specification documentation is extensive and up-to-date.

Data Standards:

This application does not use the Data Element Dictionary although it does use some DED data elements. There is a library of data names maintained by the development team to aid consistency and enforce naming conventions.

Test Data:

Extensive test data exists.

Conversion History:

No previous conversion history.

Part 4: Meeting User Requirements

Weaknesses in Meeting User Needs:

The application cannot print directly to the SO printers as some SOs would like but this is primarily due to the overload on the SO printers not the application.

One function for Miscellaneous Cost and Collection which they would like to eventually incorporate is allowing manual payment schedules for special cases to be typed directly into the system.

Users would also like to query the data on-line during the day.

Planned Improvements:

Plan to implement indexed-sequential file access in the near future.

Development Cost Ceiling:

Development Person-Months: 111
Cost (@ $3,100/month): $344,100

<< BLM SOFTWARE APPLICATION ASSESSMENT >>

Part 1: General Information

Application Name: Financial Management Edits

Contact: Alan Johnson

Age: Originally written approximately fifteen years ago

Part 2: Technical Characteristics

Operational Mode:	Batch and On-line during scheduled production
Language(s) and Size:	COBOL 33 programs 15,335 LOC
Reliability Required:	Nominal
Data Base Size:	Very High
Coding Complexity:	High
Execution Time Requirements:	Nominal
Core Storage Requirements:	Nominal
Machine Volatility:	Low
Turnaround Time:	Logic changes - Very High Compiles only - Nominal
User Population/Location:	Finance, DSC
BLM Program(s) Supported:	All financial management programs
Method of Data Entry:	Keyed and Batch

Part 3: Supportability

Language Use:

The COBOL programs are primarily within ANSI standards for COBOL 68 with only minor Honeywell specific extensions used.

Design and Development Methodology:

This application was originally developed before the current application development methodologies were implemented at BLM. The older programs within the application do not adhere to any structured programming standards.

Documentation:

Very little exists and it is out-of-date.

Data Standards:

The newer (or more recently modified) editing programs adhere to structured programming techniques and make use of the Data Dictionary and naming conventions.

Test Data:

None reported.

Conversion History:

Original programs were converted from the Burroughs computer system.

Part 4: Meeting User Requirements

Weaknesses in Meeting User Needs:

None reported.

Planned Improvements:

None reported.

Development Cost Ceiling:

Development Person-Months: 26
Cost (@ $3,100/month): $80,600

<< BLM SOFTWARE APPLICATION ASSESSMENT >>

Part 1: General Information

Application Name: Financial Management Reports

Contact: Alan Johnson

Age: Originally developed about fifteen years ago. Changes are made to programs on a regular basis, depending on reporting needs.

Part 2: Technical Characteristics

Operational Mode:	Batch, during scheduled production
Language(s) and Size:	COBOL 41 programs 38,000 LOC
Reliability Required:	Low
Data Base Size:	Very High
Coding Complexity:	Nominal to High
Execution Time Requirements:	Nominal
Core Storage Requirements:	Nominal
Machine Volatility:	Low
Turnaround Time:	Logic changes - Very High Compiles only - Nominal
User Population/Location:	Management at all levels
BLM Program(s) Supported:	Financial Management
Method of Data Entry:	From the Field Offices: from tape From DSC: key to disk

---

Part 3: Supportability

Language Use:

The application's programs are written in COBOL 68 and COBOL 74 within ANSI standards. Some Honeywell-specific extensions.

Design and Development Methodology:

This application was originally developed before the current application development methodologies were implemented at BLM. The older programs within the application do not adhere to any structured programming standards.

Documentation:

Documentation was given an overall rating of 4. The documentation for the older programs is virtually nonexistent. Only the programs written within the last five years are well documented.

Data Standards:

The newer programs adhere to structured programming techniques and make use of the Data Dictionary and naming conventions.

Test Data:

None reported.

Conversion History:

The older programs were converted form the Burroughs.

Part 4: Meeting User Requirements

Weaknesses in Meeting User Needs:

Program environment is dynamic. Changes are made prior to running reports depending on BLM reporting needs.

Planned Improvements:

None reported.

Development Cost Ceiling:

Development Person-Months: 66
Cost (@ $3,100/month): $204,600

<< BLM SOFTWARE APPLICATION ASSESSMENT >>

Part 1: General Information

Application Name: Financial Management Year-End

Contact: Alan Johnson

Age: Original skeleton programs written about fifteen years ago. Programs are tailored every year and new ones are added.

Part 2: Technical Characteristics

Operational Mode: Batch, in scheduled production

Language(s) and Size: COBOL 9 programs 4,353 LOC

Reliability Required: Low

Data Base Size: Very High

Coding Complexity: Low

Execution Time Requirements: Nominal

Core Storage Requirements: Nominal

Machine Volatility: Low

Turnaround Time: Logic changes - Very High
Compiles only - Nominal

User Population/Location: Finance, DSC, WO, management

BLM Program(s) Supported: Financial Management

Method of Data Entry: Batch

Part 3: Supportability

Language Use:

COBOL is within ANSI standards with some Honeywell extensions.

Design and Development Methodology:

This application was originally developed before the current application development methodologies were implemented at BLM. The older programs within the application do not adhere to any particular coding standards.

Documentation:

Documentation was noted as nonexistent.

Data Standards:

The newer programs adhere to structured programming techniques and make use of the Data Dictionary and naming conventions.

Test Data:

None reported.

Conversion History:

Older programs were converted from the Burroughs.

Part 4: Meeting User Requirements

Weaknesses in Meeting User Needs:

No problems mentioned. Over a period of approximately five workmonths, programs are tailored according to management needs.

Planned Improvements:

None reported.

Development Cost Ceiling:

Development Person-Months: 10
Cost (@ $3,100/month): $31,000

<< BLM SOFTWARE APPLICATION ASSESSMENT >>

Part 1: General Information

Application Name:	Inventory Data System (IDS)
Contact:	Bob Wagner (user rep), Ron Baker (programmer)
Age:	Approximately two years old

Part 2: Technical Characteristics

Operational Mode:	Runs on user request and includes both batch and timeshare components
Language(s) and Size:	COBOL 16 programs 30,000 LOC ASPEN/2 3 programs 400 LOC
Reliability Required:	Nominal to High
Data Base Size:	Very high ratio of data to LOC (approx 5,564 ch/LOC)
Complexity:	Low
Execution Time Requirements:	Nominal, less than 50% of CPU time available
Core Storage Requirements:	Nominal, less than 50% of core available
Machine Volatility:	Low
Turnaround Time:	Low for minor changes
User Population/Location:	Used primarily at Resource Area Offices
BLM Program(s) Supported:	Range Program
Method of Data Entry:	Keyed at any field office

Part 3: Supportability

Language Use:

COBOL does not conform entirely to ANSI standards and portions of the application code will have to be rewritten if this application were converted.

Design and Development Methodology:

Code is structured and modular. No data flow diagrams exist.

Documentation:

Program documentation exists for the Ecological Site Inventory System which this application is replacing. For IDS, there is no formal documentation at this time -- just record and print layouts. Documentation is currently being compiled.

Data Standards:

The Data Element Dictionary and naming conventions were used in developing this application.

Test Data:

Test data is available for testing 100% of the program logic. There are not separate test and production versions of the application code.

Conversion History:

Vegetation data for this application was converted from the Ecological Site Inventory Application.

Part 4: Meeting User Requirements

Weaknesses in Meeting User Needs:

There are no outstanding change requests and no known functions required by the users which are not available.

Planned Improvements:

In the long term, this application will be tied to GIS. These programs are still in the development stage.

Development Cost Ceiling:

Development Person-Months: 71
Cost (@ $3,100/month): $220,100

<< BLM SOFTWARE APPLICATION ASSESSMENT >>

Part 1: General Information

Application Name: Lease Management

Contact: Tim Tafoya

Age: Approximately thirteen years.

Part 2: Technical Characteristics

Operational Mode:	Batch		
Language(s) and Size:	COBOL	15 programs	15,000 LOC
Reliability Required:	Very Low		
Data Base Size:	Very High		
Coding Complexity:	Nominal		
Execution Time Requirements:	Nominal		
Core Storage Requirements:	Nominal		
Machine Volatility:	Low		
Turnaround Time:	Very High		
User Population/Location:	Field Offices		
BLM Program(s) Supported:	LM 105, 106		
Method of Data Entry:	Keyed		

Part 3: Supportability

Language Use:

COBOL is within ANSI standards. Currently being rewritten in latest version of COBOL (74) and will be within ANSI standards.

Design and Development Methodology:

The design and code were developed using structured coding techniques.

Documentation:

Documentation is being written as the new code is written. Thus will be up-to-date, and it is said, complete.

Data Standards:

Uses the Data Element Dictionary and consistent naming conventions.

Test Data:

Yes. 90% of the program logic is tested by the test data.

Conversion History:

Original application was converted from the Burroughs computer system.

Part 4: Meeting User Requirements

Weaknesses in Meeting User Needs:

Users would like to have more control of the system and would like to be able to print reports on-site.

Planned Improvements:

Currently rewriting application in the latest version of COBOL, COBOL 74.

Development Cost Ceiling:

Development Person-Months: 43
Cost (@ $3,100/month): $133,300

<< BLM SOFTWARE APPLICATION ASSESSMENT >>

Part 1: General Information

Application Name: Material Sales

Contact: Dave Alee

Age: Some parts are over ten years old but most are less than 3

Part 2: Technical Characteristics

Operational Mode: Batch, on-line, and timeshare

Language(s) and Size:

COBOL	10 programs	44000 LOC
BASIC	15 programs	10000 LOC
DM-IV	7 programs	2000 LOC
TEX	2 programs	140 LOC
CRUNS	10 programs	200 LOC

Reliability Required: Low

Data Base Size: High

Coding Complexity: Very low

Execution Time Requirements: Nominal

Core Storage Requirements: Nominal

Machine Volatility: Low

Turnaround Time: Nominal

User Population/Location: DO and RAs

BLM Program(s) Supported: Timber, vegetal, and minerals

Method of Data Entry: on-line

Part 3: Supportability

Language Use:

The Materials Sales Application uses 4 vendor-specific languages (DM-IV, ASPEN/2, TEX, and CRUNS). COBOL and BASIC are primarily within ANSI standards.

Design and Development Methodology:

Documentation:

Documentation was rated overall as poor. What little is said to exist is reported to be out-of-date with the exception of the user documentation (which was rated as usable and up-to-date, but not very complete).

Data Standards:

Test Data:

None reported.

Conversion History:

No previous conversion history.

Part 4: Meeting User Requirements

Weaknesses in Meeting User Needs:

None reported.

Planned Improvements:

None reported.

Development Cost Ceiling:

Development Person-Months: 91
Cost (@ $3,100/month): $282,100

<< BLM SOFTWARE APPLICATION ASSESSMENT >>

Part 1: General Information

Application Name: Operating Budget System

Contact: Marvin Gutweto

Age: Approximately three years old

Part 2: Technical Characteristics

Operational Mode:	Runs in scheduled production and includes both batch and timeshare components.
Language(s) and Size:	COBOL 21 programs 7700 LOC DEF II 4 programs 300 LOC
Reliability Required:	Low
Data Base Size:	High ratio of data to LOC (474 ch/LOC)
Coding Complexity:	Low
Execution Time Requirements:	Nominal, less than 50% of CPU time available.
Core Storage Requirements:	Nominal, less than 50% of core available.
Machine Volatility:	Low
Turnaround Time:	Nominal, within four hours
User Population/Location:	Used primarily at the State Offices with some reports possibly distributed to the District level.
BLM Program(s) Supported:	All programs
Method of Data Entry:	On-line data entry by State Office staff

---

Part 3: Supportability

Language Use:

The Operating Budget System uses two languages: COBOL 74 and DEF II. The COBOL is within ANSI standards and has no device or machine dependent code. The DEF II is Honeywell specific and non-standard. It is used on the Level 6s to format input data and call the COBOL programs running on the DPS-8/70.

If this application were converted, the DEF II portion would have to be rewritten in a language suitable to the target environment.

Design and Development Methodology:

The COBOL programs conform to BLM's structured design and coding standards and are modular. Most but not all of the documentation was developed in a document-as-you-go fashion.

Documentation:

Overall documentation is average to weak (4). Documentation is very strong in functional specification and user documentation but weak in the program description and program specification areas.

Data Standards:

The applications do not use the Data Element Dictionary or any other formal data standards. However, the programs were all written by the same person and use consistent naming conventions.

Test Data:

Test data is generated from live data rather than maintained in a benchmark form of library. There are separate test and production versions of the application code.

Conversion History:

No previous conversion history.

Part 4: Meeting User Requirements

Weaknesses in Meeting User Needs:

There are no outstanding change requests and no known functions required by the users which are not available.

Planned Improvements:

None reported.

Development Cost Ceiling:

Development Person-Months: 16
Cost (@ $3,100/month): $49,600

<< BLM SOFTWARE APPLICATION ASSESSMENT >>

Part 1: General Information

Application Name: Program Management System

Contact: Rick Graham

Age: 2 years

Part 2: Technical Characteristics

Operational Mode: Batch in production

Language(s) and Size: COBOL 10 programs 13,061 LOC

Reliability Required: Very low

Data Base Size: Very high

Coding Complexity: Nominal (matrix processing)

Execution Time Requirements: Nominal

Core Storage Requirements: Nominal

Machine Volatility: Low

Turnaround Time: High

User Population/Location: RA, DO, and SO level reports

BLM Program(s) Supported: All programs at the management level

Method of Data Entry: Key to disk

Part 3: Supportability

Language Use:

COBOL is within ANSI standards and machine independent. There is no use of machine dependent languages.

Design and Development Methodology:

Developed using structure design and coding techniques. The code conforms to the Computer Applications Division Standards for DSC and was documented as it was developed.

Documentation:

Documentation is strong (8 on a 10 point scale) in both completeness and being up to date.

Data Standards:

Uses the Data Element Dictionary in some programs but not all. It has consistent naming conventions within the application and is consistent with the DED where possible.

Test Data:

No test data.

Conversion History:

No previous conversion history.

Part 4: Meeting User Requirements

Weaknesses in Meeting User Needs:

There are no functional areas which do not meet user needs. There are two outstanding change requests but they are minor and computational and will not change overall function.

Planned Improvements:

The application generated excessive detail in its reports at first. It has been modified to calculate the detail but not display it. If the users find they do not need the detail, this logic will be removed.

Development Cost Ceiling:

Development Person-Months: 25
Cost (@ $3,100/month): $77,500

<< BLM SOFTWARE APPLICATION ASSESSMENT >>

Part 1: General Information

Application Name:	Property Management - Automated Personal Property System (APPS)
Contact:	Shirley Krebs (user rep), Fred Robinson (programmer)
Age:	Approximately two years old

Part 2: Technical Characteristics

Operational Mode:	This application is executed by the user directly and has both batch and timeshare components
Language(s) and Size:	COBOL 40 programs 42,180 LOC
Reliability Required:	Very low
Data Base Size:	High ratio of data to LOC (approx. 452 ch/LOC)
Complexity:	High
Execution Time Requirements:	Nominal, less than 50% of CPU time available
Core Storage Requirements:	Nominal, less than 50% of core available
Machine Volatility:	Low
Turnaround Time:	Very high. Most changes are involved and take a bit of time (includes making changes and thoroughly testing)
User Population/Location:	Accessed by State and District Offices
BLM Program(s) Supported:	Financial Management
Method of Data Entry:	On-line or keyed at DSC from input received from field offices

---

Part 3: Supportability

Language Use:

COBOL is compliant with ANSI standards.

Design and Development Methodology:

Code is estimated to be 75-80% structured and modular. 20-25% is aid to need work. No data flow diagrams are available.

Documentation:

Although all types of documentation exists, it is said to be very weak (4).

Data Standards:

APPS programs use a Data Element Dictionary and standard naming conventions.

Test Data:

Test data exists for testing 100% of the program logic. This test data was last updated January, 1987. There are separate test and production versions of this application.

Conversion History:

The original property management system ran on the Burroughs. The application was rewritten to run on the Honeywell.

Part 4: Meeting User Requirements

Weaknesses in Meeting User Needs:

User request for faster input is currently being addressed.

Planned Improvements:

Property numbers will be barcoded so input can be accomplished more quickly using barcode readers.

Development Cost Ceiling:

Development Person-Months: 91
Cost (@ $3,100/month): $282,100

<< BLM SOFTWARE APPLICATION ASSESSMENT >>

Part 1: General Information

Application Name: Reimbursable Billing

Contact: Alan Johnson

Age: Approximately four years

Part 2: Technical Characteristics

Operational Mode:	Batch, run in scheduled production
Language(s) and Size:	COBOL 24 programs 27,072 LOC
Reliability Required:	Low
Data Base Size:	Nominal
Coding Complexity:	Nominal
Execution Time Requirements:	Nominal
Core Storage Requirements:	Nominal
Machine Volatility:	Low
Turnaround Time:	Logic changes - Very High Compiles only - Nominal
User Population/Location:	Primarily Finance, DSC, SOs
BLM Program(s) Supported:	SIBAC
Method of Data Entry:	Batch

Part 3: Supportability

Language Use:

COBOL application code conforms to ANSI standards with minor Honeywell specific extensions.

Design and Development Methodology:

Code is fairly well structured.

Documentation:

Overall, documentation was rated fair (an extensive amount was said to be incomplete and out-of-date, but usable).

Data Standards:

Programs generally adhere to structured programming techniques and make use of the Data Dictionary and naming conventions.

Test Data:

None reported.

Conversion History:

No previous conversion history.

Part 4: Meeting User Requirements

Weaknesses in Meeting User Needs:

None reported.

Planned Improvements:

None reported.

Development Cost Ceiling:

Development Person-Months: 93
Cost (@ $3,100/month): $288,300

<< BLM SOFTWARE APPLICATION ASSESSMENT >>

Part 1: General Information

Application Name: Summer Hire Program

Contact: Jim Sanders

Age: Approximately six years old

Part 2: Technical Characteristics

Operational Mode: Executed by user directly and includes batch components

Language(s) and Size: COBOL 4 programs 8,000 LOC

Reliability Required: Low

Data Base Size: Very high ratio of data to LOC (2,475 ch/LOC)

Complexity: Nominal

Execution Time Requirements: Nominal, less than 50% of CPU time available

Core Storage Requirements: Nominal, less than 50% of core available

Machine Volatility: Low

Turnaround Time: High

User Population/Location: Denver Service Center (DSC)

BLM Program(s) Supported: Summer Hiring

Method of Data Entry: Batch process, key-to-disk at DSC

Part 3: Supportability

Language Use:

The Summer Hire Application uses COBOL and is within ANSI standards and has no device or machine dependent code.

Design and Development Methodology:

The COBOL programs are estimated to be approximately 80% structured and are said to have poor modularity. System flowcharts are available, but no data flow diagrams. An up-to-date set of application documentation is not available.

Documentation:

Program and User documentation exists for this application and was rated high (9). All changes made to the application have been documented, but the overall documentation has never been "cleaned up" (e.g., items that no longer hold true are still included). No other documentation exists.

Data Standards:

This application does not use the Data Element Dictionary or naming conventions.

Test Data:

Test data exists for this application which tests approximately 85% of the programming logic. This test data was last updated in November, 1986. There are separate test and production versions of the application code.

Conversion History:

No previous conversion history.

Part 4: Meeting User Requirements

Weaknesses in Meeting User Needs:

There are no outstanding change requests and no known functions required by the users which are not available. Formulas will need to be changed once in a while, when, for example, qualifications change or additional items are required on reports.

Planned Improvements:

None reported.

Development Cost Ceiling:

Development Person-Months: 16
Cost (@ $3,100/month): $49,600

<< BLM SOFTWARE APPLICATION ASSESSMENT >>

Part 1: General Information

Application Name: USFS Forest Inventory

Contact: Bill Williams

Age: 5-7 years

Part 2: Technical Characteristics

Operational Mode: Batch and timeshare operation by users directly

Language(s) and Size:

FORTRAN IV	5 programs	10,000 LOC
TEX	10 programs	200 LOC

Reliability Required: Very low

Data Base Size: Very high

Coding Complexity: Nominal (Matrix processing)

Execution Time Requirements: Nominal

Core Storage Requirements: Nominal

Machine Volatility: Low

Turnaround Time: Nominal

User Population/Location: DO or RA foresters; DSC enters data from forms

BLM Program(s) Supported: Forestry planning

Method of Data Entry: Data keyed at DSC

Part 3: Supportability

Language Use:

The application code was written by the US Forest Service in FORTRAN IV and is within ANSI standards although FORTRAN IV is not the most current version of FORTRAN. The FORTRAN has no machine dependent code but the supporting TEX programs written by BLM are Honeywell dependent.

Design and Development Methodology:

The code is modular and follows many of the structured programming guidelines but not consistently and there is some "spaghetti" code. However, BLM does not maintain the code so the impact of unstructured code is minimal. USFS occasionally sends BLM a tape with updates to the application code but the modules are pretty stable from a functional perspective.

Documentation:

Most of the development specifications (e.g. functional specs, system specs, and program specs) have been retained by USFS. BLM has primarily operational and user documentation.

Data Standards:

These applications are internally consistent but do not comply with BLM data elements or use the BLM Data Element Dictionary since they were developed outside of BLM.

Test Data:

Complete set of test data is available.

Conversion History:

No previous conversion history.

Part 4: Meeting User Requirements

Weaknesses in Meeting User Needs:

Users have complained about slow response time, communication problems, and slow turnaround but have no complaints about application functionality.

Planned Improvements:

There are no outstanding change requests for these applications and it has been several years since the USFS provided any updates. No improvements are planned at this time.

Development Cost Ceiling:

Development Person-Months: 18
Cost (@ $3,100/month): $55,800

<< BLM SOFTWARE APPLICATION ASSESSMENT >>

Part 1: General Information

Application Name: Waterpower System

Contact: Alan Johnson

Age: Five years at BLM but acquired from the Army Corps of Engineers which may have had it for ten or more years.

Part 2: Technical Characteristics

Operational Mode: Remote Job Entry (RJE) by users

Language(s) and Size:

FORTRAN	25 programs	2,000,000 LOC
TEX	3 programs	10,000 LOC

Approximately 60 CRUNS and DRUNS, unknown LOC

Reliability Required: Very low

Data Base Size: Nominal

Coding Complexity: Very high

Execution Time Requirements: Nominal

Core Storage Requirements: Nominal, although one program, HEC5, requires 190K

Machine Volatility: Low

Turnaround Time: Nominal

User Population/Location: Resource Area staff in primarily OSO, CaSO, and CSO

BLM Program(s) Supported: Waterpower program

Method of Data Entry: Keyed by users

Part 3: Supportability

Language Use:

Written primarily in FORTRAN 77 which conforms to ANSI standards and has no machine specific code. However, these applications also make extensive use of TEX, a Honeywell-specific text manipulator; CRUNS, which execute programs as background mode on Honeywell; and DRUNS, which allow deferred program execution in the Honeywell environment.

Design and Development Methodology:

These programs represent more of a tool kit for hydrology engineers then an integrated system. The FORTRAN programs are not modular and do not use structured techniques however their functionality is extremely stable and no changes are anticipated so the impact of this lack of formal methodology is minimal.

Documentation:

Documentation is rated at 4 on a scale of 1-10. Very highly rated functional requirements and system specifications but weak user and program documentation. The applications rely heavily on the user to already know what he's doing.

Data Standards:

Consistent naming conventions are used although some variable names are cryptic. If any standards were used, they were Army Corps of Engineer standards not BLM.

Test Data:

BLM maintains test data separate from production data which tests approximately 60% of the logic paths.

Conversion History:

No previous conversion history.

Part 4: Meeting User Requirements

Weaknesses in Meeting User Needs:

Users would like data entry screens. They also complain that some of the

applications take a long time to execute. This is frequently due to the large amount of core required by some of the programs, HEC5 in particular. During busy periods, these programs can wait for hours for sufficient core to be available for them to execute.

Planned Improvements:

The Corps of Engineers is moving some of these programs onto microcomputers and BLM hopes to get copies of these programs. BLM is also planning to develop data entry screens for the users.

Development Cost Ceiling:

Development Person-Months: 214
Cost (@ $3,100/month): $663,400

<< BLM SOFTWARE APPLICATION ASSESSMENT >>

Part 1: General Information

Application Name: Wildfire Reporting System

Contact: Helen Costello (programmer)

Age: Approximately years old

Part 2: Technical Characteristics

Operational Mode:	Executed by user with both batch and timeshare components		
Language(s) and Size:	COBOL	7 programs	19,760 LOC
	TEX	12 programs	600 LOC
Reliability Required:	Low		
Data Base Size:	High ratio of data to LOC (137 chs/LOC)		
Complexity:	Nominal to High		
Execution Time Requirements:	Nominal		
Core Storage Requirements:	Nominal		
Machine Volatility:	Low		
Turnaround Time:	Nominal		
User Population/Location:	BIFC		
BLM Program(s) Supported:	Fire program		
Method of Data Entry:	Keyed at BIFC, Alaska may do some of their own keying		

Part 3: Supportability

Language Use:

The Wildfire Reporting System uses ANSI standard COBOL 74. The ASPEN/2 and TEX programs are machine specific and would require a total rewrite if the target environment is not compatible with the existing environment.

Design and Development Methodology:

The application's programs are said to be somewhat structured -- the code is basically in compliance with BLM design and coding standards, but could be more structured. Modules range from 10 to approximately 100 LOC.

Documentation:

Overall documentation is rated as average (5). Functional Requirements, Program Documentation, and User documentation are rated as very good (9), but other types (e.g., database and programs specifications, etc.) were rated low (4's).

Data Standards:

A Data Element Dictionary is used in this application, but standard naming conventions are not.

Test Data:

Test data exists which tests approximately 10% of the program logic. Separate test and production versions of the Wildfire Reporting System are used.

Conversion History:

No previous conversion history.

Part 4: Meeting User Requirements

Weaknesses in Meeting User Needs:

In FY 86, there were 15 or so minor changes made. Currently, there are no outstanding change requests and no known functions required by the users which are not available.

Planned Improvements:

In the process of converting State and District Office codes from numeric to alphanumeric.

Development Cost Ceiling:

Development Person-Months: 28
Cost (@ $3,100/month): $86,800

<< BLM SOFTWARE APPLICATION ASSESSMENT >>

Part 1: General Information

Application Name: Wildlife Information System

Contact: Kurtis Ballantyne (user rep), Bob Seeley (programmer)

Age: Approximately six years, newest version - approximately three years

---

Part 2: Technical Characteristics

Operational Mode: Executed by user directly and includes both batch and timeshare components

Language(s) and Size:

COBOL	18 programs	36,000 LOC
TEX	9 programs	10,000 LOC
ASPEN/2	9 programs	XX LOC

Reliability Required: Very low

Data Base Size: Very high ratio of data to LOC (approx. 3,218 ch/LOC)

Complexity: Low

Execution Time Requirements: Nominal, less than 50% of CPU time available

Core Storage Requirements: Nominal, less than 50% of core available

Machine Volatility: Low

Turnaround Time: Very low

User Population/Location: Used primarily at the Resource Area and District Offices with summary information used by State and Washington Offices

BLM Program(s) Supported: Threatened and Endangered Species, Habitat Management

Method of Data Entry: Keyed at the Resource Area Offices

Part 3: Supportability

Language Use:

The Wildlife Information System application uses three languages: COBOL 74, TEX, and ASPEN/2. The COBOL is within ANSI standards and has no device or machine dependent code. The ASPEN/2 and TEX portions are machine dependent and non-standard.

If this application were converted, the TEX programs would have to be rewritten, and the ASPEN/2 programs rewritten unless ASPEN/2 could run on the new system.

Design and Development Methodology:

The programs are structured and modular.

Documentation:

Though not very complete, an operations manual and user documentation exists for this applications. Data flow diagrams are available for the earlier version of the application, but no updated ones are. Overall documentation was rated a 5.

Data Standards:

This application uses the Data Element Dictionary and uses consistent naming conventions.

Test Data:

No test data exists for this application.

Conversion History:

No previous conversion history.

Part 4: Meeting User Requirements

Weaknesses in Meeting User Needs:

There are no outstanding change requests and no known functions required by the users which are not available.

Planned Improvements:

None reported.

Development Cost Ceiling:

Development Person-Months: 96
Cost (@ $3,100/month): $297,600

APPENDIX B

COCOMO Cost Estimates

For Redevelopment of Bureau-wide Applications

B.1 INTRODUCTION

This Appendix contains the results of running the COCOMO model on the nineteen applications examined in Chapter 4 of this document. The second section of this appendix provides an overview of the COCOMO cost estimating model and its underlying assumptions. The model was described briefly in Chapter 2 of this document. For the readers' understanding, we have provided a more detailed description in this appendix. The third section describes the COCOMO spreadsheet components so the reader may better understand the output.

B.2 OVERVIEW OF THE COCOMO MODEL

Figure B-1 contains the output generated from executing the COCOMO cost model using the application attributes obtained in the surveys contained in Appendix A. The model estimates the development man-months required for a given software project.

The COCOMO model for estimating software development cost was developed by Barry Boehm of TRW. There are three steps involved in using the model:

1. Determine the size of the applications in terms of the number of lines of code.

2. Select the Development Mode that best describes how the project was developed (Organic, Semi-detached, or Embedded[1]) so the model "knows" which equations to use to generate the results.

3. Rate the application according to fifteen "cost drivers". These drivers are:

   - Required Software Reliability (RELY)
   - Data Base Size (DATA)
   - Product Complexity (CPLX)

Figure B-1
COCOMO RESULTS

EFFORT RATINGS BY COST DRIVER

1=very low 2=low 3=nominal 4=high 5=very high 6=extra high

Component	DSI	EDSI	AAF	RELY	DATA	CPLX	TIME	STOR	VIRT	TURN	ACAP	AEXP	PCAP	VEXP	LEXP	MODP	TOOL	SCED	EAF	Nominal Person-Months	Devlpmt Person-Months
Operating Budget	8,000	8,000	0	2	4	2	3	3	2	3	3	3	3	3	4	4	4	3	0.55	28	16
Oil & Gas Field Note	19,782	19,782	0	3	5	1	3	3	2	3	3	3	3	3	4	4	4	3	0.56	73	41
Mineral Lease System	80,600	80,600	0	1	3	5	3	3	2	3	3	3	3	3	4	4	4	3	0.67	321	214
Forest Inventor	8,268	8,268	0	1	5	3	3	3	2	3	3	3	3	3	4	4	4	3	0.60	29	18
Program Management	11,063	11,063	0	1	5	3	3	3	2	4	3	3	3	3	4	4	4	3	0.64	40	25
Trust Fund	4,353	4,353	0	2	5	2	3	3	2	3	3	3	3	3	4	4	4	3	0.64	15	10
Reports	38,000	38,000	0	2	5	4	3	3	2	3	3	3	3	3	4	4	4	3	0.46	146	66
Bids	15,335	15,335	0	3	5	4	3	3	2	3	3	3	3	3	4	4	4	3	0.46	56	26
Responsable Filing	27,072	27,072	0	2	3	3	3	3	2	3	3	3	3	3	4	4	4	3	0.91	102	93
Lease Management	15,000	15,000	0	1	5	3	3	3	2	5	3	3	3	3	4	4	4	3	0.78	55	43
Motor Fleet Maint.	106,000	106,000	0	1	3	3	3	3	2	5	3	3	3	3	4	4	4	3	0.68	428	292
Monies to Treasur.	45,000	45,000	0	4	4	1	3	3	2	4	3	3	3	3	4	4	4	3	0.64	174	111
Material Sales	51,278	51,278	0	2	4	1	3	3	2	3	3	3	3	3	4	4	4	3	0.46	200	91
Wildfire Reporting	16,688	16,688	0	2	4	1	3	3	2	3	3	3	3	3	4	4	4	3	0.46	61	28
Accept a-horse	35,750	35,750	0	3	5	4	3	3	2	3	3	3	3	3	4	4	4	3	0.91	137	125
Inventory Data	24,470	24,470	0	4	5	2	3	3	2	3	3	3	3	3	4	4	4	3	0.78	92	71
Personal Property	35,050	35,050	0	1	4	4	3	3	2	4	3	3	3	3	4	4	4	3	0.68	134	91
Wildlife Information	36,800	36,800	0	1	5	2	3	3	2	3	3	3	3	3	4	4	4	3	0.68	141	96
Summer Hire	6,480	6,480	0	2	5	3	3	3	2	3	3	3	3	3	4	4	4	3	0.70	23	16
Total		584,989																	Total	2,257	1,473
EDSI Nominal Person-Month		259																	Scheduled Times		40.0 months
																			EDSI/Devlpmt Person-Month		397

- Execution Time Constraint (TIME)
- Main Storage Constraint (STOR)
- Virtual Machine Volatility[2] (VIRT)
- Computer Turnaround Time (TURN)
- Analyst Capability (ACAP)
- Applications Experience (AEXP)
- Programmer Capability (PCAP)
- Virtual Machine Experience (VEXP)
- Programming Language Experience (LEXP)
- Modern Programming Practices Employed (MODP)
- Use of Software Tools (TOOL)
- Required Development Schedule (SCED)

The following assumptions underlie the COCOMO model:

- The software development process is composed of ten phases:

  1 - Feasibility
  2 - Organization and Planning
  3 - Software Requirements Specification
  4 - Product Design Specifications
  5 - Detailed Design Specifications
  6 - Code
  7 - Unit Test
  8 - Integration and Test
  9 - Acceptance Test
  10- Operation and Maintenance

- The development period covered by the Model's cost estimate for this project is defined as the beginning of the product design phase (4) through the end of the integration and test phase (8).

- The COCOMO Model estimates cover only those software development activities listed in Figure B-1. Some

efforts which take place during the development period (e.g., user training) are not included.

- For the activities listed in Figure B-1, the (COCOMO) Model estimates include all direct labor charges. For example, project managers and program librarians will be included. Indirect labor charges, however, are excluded (e.g., computer center operators, secretaries, higher management).

- A COCOMO man-month is equivalent to 152 hours of working time.

- The Model estimates assume that the project will be well managed (i.e., nonproductive slack time is kept at a minimum).

- The Model assumes that the requirements specification will not undergo any substantial changes after the requirements phase has ended. Some refinements and reinterpretations are assumed, but any significant modifications should be covered by running a revised cost estimate.

- In order not to apply a judgement factor, AMS assigned average ratings to all of the personnel attribute factors.

B.3 Explanation of Output

This section explains the components of the results of the COCOMO contained in Figure B-1.

Column 1:

This column contains the name of the application for which the model is estimating development costs.

Column 2:

DSI stands for Delivered Source Instructions. This is generally equal to the lines of code in an application.

Column 3:

EDSI stands for Equivalent Delivered Source Instructions. This would be different from DSI if the new application is being developed from an older, similar application.

Column 4:

If starting with programs written for a similar project that will be modified for the new application, an Adaptive Adjustment Factor (AAF) must also be determined. Since we are estimated costs for total redevelopment, starting from the product design phase, this factor was given a value of zero (0) for all applications.

Column 5 through 19:

This columns contains the values given to the fifteen cost drivers as listed in section B.2 above.

Column 20:

EAF is an acronym for Effort Adjustment Factor. This is generated by the model. COCOMO maintains a table of development "effort multipliers" which are keyed to the cost driver attribute ratings for the application. For example, an application with a Software Reliability Required (RELY) rating of 3 (nominal) would have a

corresponding effort multiplier of 1.00. The EAF is a product of the effort multipliers.

Column 21:

Nominal Person-Months is the number of man-months required to redevelop the application without taking into consideration the adjustment factor (EAF) generated from incorporating the effects of the fifteen cost drivers.

Column 22:

Development Person-Months is the number of man-months required to redevelop the application incorporating the effects of the fifteen cost drivers.

Summary Lines:

EDSI/Nominal Person-Month:

Without taking into consideration the fifteen cost drivers which affect the project's productivity, it is estimated that, overall, ADP support staff can produce 259 lines of code per month.

EDSI/Development Person-Month:

Taking into consideration the fifteen cost drivers, it is estimated that, overall, ADP support staff can produce 397 lines of code per month. Adjusting the project for the fifteen cost drivers (versus the Nominal Person-Month which is equivalent to setting the value of all cost drivers to 3 (nominal)) indicates that productivity for the project is greater than "nominal".

Figure B-2 lists the resources, in dollar amounts, for the redevelopment of the applications. BLM management provided AMS with a figure of $3,100 per man-month, and this figure was used to compute the dollar estimate of redeveloping Bureau-wide applications.

Figure B-2
ESTIMATED COSTS

These costs are based on the Development Person-Months estimated by the COCOMO model and a rate of $3,100 per man-month (this figure was provided by BLM).

Development Application	Man-Months	Cost
Adopt-A-Horse	125	$387,500
Automated Fleet Management	292	$905,200
Cadastral Survey Field Notes	41	$127,100
Checks to Treasury	111	$344,100
Financial Management Edits	26	$80,600
Financial Management Reporting	66	$204,600
Financial Management Year-End	10	$31,000
Inventory Data System	71	$220,100
Lease Management	43	$133,300
Material Sales	91	$282,100
Operating Budget	16	$49,600
Program Management	25	$77,500
Property Management	91	$282,100
Reimbursable Billing	93	$288,300
Summer Hire System	16	$49,600
USFS Forest Inventory	18	$55,800
Waterpower System	214	$663,400
Wildfire Reporting	28	$86,800
Wildlife Information System	96	$297,600

NOTES

[1] Boehm defines three discrete modes which relate to the degree of overhead and the number of constraints the project is under. These Include:

Organic: The people connected with the project have a thorough understanding of the software application objectives. There is:

- extensive experience in working with related software systems
- a basic need for software conformance with requirements
- a basic need for software conformance with external interfaces
- some concurrent development of new operational procedures
- minimal need for innovative data processing algorithms or architectures
- a low premium on early completion
- typically fewer than 50,00 DSI

Semidetached: The people connected with the project have a considerable understanding of the software application objectives. There is:

- considerable experience in working with related software systems
- considerable need for software conformance with requirements
- considerable need for software conformance with external interfaces
- moderate concurrent development of new operational procedures
- some need for innovative data processing algorithms or architectures

- a medium premium on early completion
- typically fewer than 300,000 DSI

Embedded: The people connected with the project have a general understanding of the software application objectives. There is:

- moderate experience in working with related software systems
- full need for software conformance with requirements
- full need for software conformance with external interfaces
- extensive concurrent development of new operational procedures
- considerable need for innovative data processing algorithms or architectures
- a high premium on early completion

[2] The virtual machine is the complex of hardware and software (operating system, DBMS, etc.) that a given application calls on to accomplish its tasks. Volatility, here, refers to the degree of change in hardware and software.

APPENDIX D

COCOMO Cost Estimates

For Redevelopment of Bureau-wide Applications

D.1 INTRODUCTION

This Appendix contains the results of running the COCOMO model on the nineteen applications examined in Chapter 4 of this document. The second section of this appendix provides an overview of the COCOMO cost estimating model and its underlying assumptions. The model was described briefly in Chapter 2 of this document. For the readers' understanding, we have provided a more detailed description in this appendix. The third section describes the COCOMO spreadsheet components so the reader may better understand the output.

D.2 OVERVIEW OF THE COCOMO MODEL

Figure D-1 contains the output generated from executing the COCOMO cost model using the application attributes obtained in the surveys contained in Appendix C. The model estimates the development man-months required for a given software project.

The COCOMO model for estimating software development cost was developed by Barry Boehm of TRW. There are three steps involved in using the model:

1. Determine the size of the applications in terms of the number of lines of code.

2. Select the Development Mode that best describes how the project was developed (Organic, Semi-detached, or Embedded[1]) so the model "knows" which equations to use to generate the results.

3. Rate the application according to fifteen "cost drivers". These drivers are:

    - Required Software Reliability (RELY)
    - Data Base Size (DATA)
    - Product Complexity (CPLX)

Figure D-1
COCOMO Results

EFFORT RATINGS BY COST DRIVER

1 = very low 2 = low 3 = nominal 4 = high 5 = very high

Component	DSI	EDSI	AAF	RELY	DATA	CPLX	TIME	STOR	VIRT	TURN	ACAP	AEXP	PCAP	VEXP	LEXP	MODP	TOOL	SCED	EAF	Nominal Person-Months	Development Person-Months
Operating Budget	8,000	8,000	0	2	4	2	3	3	2	3	3	3	3	3	4	4	4	3	0.55	28	16
Cadastral Field Note	19,782	19,782	0	3	5	1	3	3	2	3	3	3	3	3	4	4	4	3	0.56	73	41
Waterpower System	80,600	80,600	0	1	3	5	3	3	2	3	3	3	3	3	4	4	4	3	0.67	321	214
USFS Forest Inventory	8,268	8,268	0	1	5	3	3	3	2	3	3	3	3	3	4	4	4	3	0.60	29	18
Program Management	11,063	11,063	0	1	5	3	3	3	2	4	3	3	3	3	4	4	4	3	0.64	40	25
Financial Management Year-End	4,353	4,353	0	2	5	2	3	3	2	3	3	3	3	3	4	4	4	3	0.64	15	10
Financial Management Reports	38,000	38,000	0	2	5	4	3	3	2	3	3	3	3	3	4	4	4	3	0.46	146	66
Financial Management Edits	15,335	15,335	0	3	5	4	3	3	2	3	3	3	3	3	4	4	4	3	0.46	56	26
Reimbursable Billing	27,072	27,072	0	2	3	3	3	3	2	3	3	3	3	3	4	4	4	3	0.91	102	93
Lease Management	15,000	15,000	0	1	5	3	3	3	2	5	3	3	3	3	4	4	4	3	0.78	55	43
Automated Fleet Management	106,000	106,000	0	1	3	3	3	3	2	5	3	3	3	3	4	4	4	3	0.68	428	292
Checks to Treasury	45,000	45,000	0	4	4	1	3	3	2	4	3	3	3	3	4	4	4	3	0.64	174	111
Material Sales	51,278	51,278	0	2	4	1	3	3	2	3	3	3	3	3	4	4	4	3	0.46	200	91
Wildfire Reporting	16,688	16,688	0	2	4	1	3	3	2	3	3	3	3	3	4	4	4	3	0.46	61	28
Adopt-a-Horse	35,750	35,750	0	3	5	4	3	3	2	3	3	3	3	3	4	4	4	3	0.91	137	125
Inventory Data	24,470	24,470	0	4	5	2	3	3	2	3	3	3	3	3	4	4	4	3	0.78	92	71
Personal Property	35,050	35,050	0	1	4	4	3	3	2	4	3	3	3	3	4	4	4	3	0.68	134	91
Wildlife Information	36,800	36,800	0	1	5	2	3	3	2	3	3	3	3	3	4	4	4	3	0.68	141	96
Summer Hire	6,480	6,480	0	2	5	3	3	3	2	3	3	3	3	3	4	4	4	3	0.70	23	16
TOTAL		584,989																		2,257	1,473

EDSI/Nominal Person-Month 259

Scheduled Times 40.0 months

EDSI/Development Person-Month 397

- Execution Time Constraint (TIME)
- Main Storage Constraint (STOR)
- Virtual Machine Volatility[2] (VIRT)
- Computer Turnaround Time (TURN)
- Analyst Capability (ACAP)
- Applications Experience (AEXP)
- Programmer Capability (PCAP)
- Virtual Machine Experience (VEXP)
- Programming Language Experience (LEXP)
- Modern Programming Practices Employed (MODP)
- Use of Software Tools (TOOL)
- Required Development Schedule (SCED)

The following assumptions underlie the COCOMO model:

- The software development process is composed of ten phases:

  1 - Feasibility
  2 - Organization and Planning
  3 - Software Requirements Specification
  4 - Product Design Specifications
  5 - Detailed Design Specifications
  6 - Code
  7 - Unit Test
  8 - Integration and Test
  9 - Acceptance Test
  10- Operation and Maintenance

- The development period covered by the Model's cost estimate for this project is defined as the beginning of the product design phase (4) through the end of the integration and test phase (8).

- The COCOMO Model estimates cover only those software development activities listed in Figure D-1. Some

efforts which take place during the development period (e.g., user training) are not included.

- For the activities listed in Figure D-1, the COCOMO Model estimates include all direct labor charges. For example, project managers and program librarians will be included. Indirect labor charges, however, are excluded (e.g., computer center operators, secretaries, higher management).

- A COCOMO man-month is equivalent to 152 hours of working time.

- The Model estimates assume that the project will be well managed (i.e., nonproductive slack time is kept at a minimum).

- The Model assumes that the requirements specification will not undergo any substantial changes after the requirements phase has ended. Some refinements and reinterpretations are assumed, but any significant modifications should be covered by running a revised cost estimate.

- In order not to apply a judgement factor, AMS assigned average ratings to all of the personnel attribute factors.

D.3 Explanation of Output

This section explains the components of the results of the COCOMO contained in Figure D-1.

Column 1:

This column contains the name of the application for which the model is estimating development costs.

Column 2:

DSI stands for Delivered Source Instructions. This is generally equal to the lines of code in an application.

Column 3:

EDSI stands for Equivalent Delivered Source Instructions. This would be different from DSI if the new application is being developed from an older, similar application.

Column 4:

If starting with programs written for a similar project that will be modified for the new application, an Adaptive Adjustment Factor (AAF) must also be determined. Since we are estimated costs for total redevelopment, starting from the product design phase, this factor was given a value of zero (0) for all applications.

Column 5 through 19:

This columns contains the values given to the fifteen cost drivers as listed in section B.2 above.

Column 20:

EAF is an acronym for Effort Adjustment Factor. This is generated by the model. COCOMO maintains a table of development "effort multipliers" which are keyed to the cost driver attribute ratings for the application. For example, an application with a Software Reliability Required (RELY) rating of 3 (nominal) would have a

corresponding effort multiplier of 1.00. The EAF is a product of the effort multipliers.

Column 21:

Nominal Person-Months is the number of man-months required to redevelop the application without taking into consideration the adjustment factor (EAF) generated from incorporating the effects of the fifteen cost drivers.

Column 22:

Development Person-Months is the number of man-months required to redevelop the application incorporating the effects of the fifteen cost drivers.

Summary Lines:

EDSI/Nominal Person-Month:

Without taking into consideration the fifteen cost drivers which affect the project's productivity, it is estimated that, overall, ADP support staff can produce 259 lines of code per month.

EDSI/Development Person-Month:

Taking into consideration the fifteen cost drivers, it is estimated that, overall, ADP support staff can produce 397 lines of code per month. Adjusting the project for the fifteen cost drivers (versus the Nominal Person-Month which is equivalent to setting the value of all cost drivers to 3 (nominal)) indicates that productivity for the project is greater than "nominal".

Figure D-2 lists the resources, in dollar amounts, for the redevelopment of the applications. BLM management provided AMS with a figure of $3,100 per man-month, and this figure was used to compute the dollar estimate of redeveloping Bureau-wide applications.

Figure D-2
ESTIMATED COSTS

These costs are based on the Development Person-Months estimated by the COCOMO model and a rate of $3,100 per man-month (this figure was provided by BLM).

Development Application	Man-Months	Cost
Adopt-A-Horse	125	$387,500
Automated Fleet Management	292	$905,200
Cadastral Survey Field Notes	41	$127,100
Checks to Treasury	111	$344,100
Financial Management Edits	26	$80,600
Financial Management Reporting	66	$204,600
Financial Management Year-End	10	$31,000
Inventory Data System	71	$220,100
Lease Management	43	$133,300
Material Sales	91	$282,100
Operating Budget	16	$49,600
Program Management	25	$77,500
Property Management	91	$282,100
Reimbursable Billing	93	$288,300
Summer Hire System	16	$49,600
USFS Forest Inventory	18	$55,800
Waterpower System	214	$663,400
Wildfire Reporting	28	$86,800
Wildlife Information System	96	$297,600

NOTES

[1] Boehm defines three discrete modes which relate to the degree of overhead and the number of constraints the project is under. These Include:

Organic: The people connected with the project have a thorough understanding of the software application objectives. There is:

- extensive experience in working with related software systems
- a basic need for software conformance with requirements
- a basic need for software conformance with external interfaces
- some concurrent development of new operational procedures
- minimal need for innovative data processing algorithms or architectures
- a low premium on early completion
- typically fewer than 50,00 DSI

Semidetached: The people connected with the project have a considerable understanding of the software application objectives. There is:

- considerable experience in working with related software systems
- considerable need for software conformance with requirements
- considerable need for software conformance with external interfaces
- moderate concurrent development of new operational procedures
- some need for innovative data processing algorithms or architectures

- a medium premium on early completion
- typically fewer than 300,000 DSI

Embedded: The people connected with the project have a general understanding of the software application objectives. There is:

- moderate experience in working with related software systems
- full need for software conformance with requirements
- full need for software conformance with external interfaces
- extensive concurrent development of new operational procedures
- considerable need for innovative data processing algorithms or architectures
- a high premium on early completion

[2] The virtual machine is the complex of hardware and software (operating system, DBMS, etc.) that a given application calls on to accomplish its tasks. Volatility, here, refers to the degree of change in hardware and software.

APPENDIX E

REFERENCES FOR THE BLM SOFTWARE ASSESSMENT

REFERENCES

Boehm, Barry W. Software Engineering Techniques, Prentice-Hall Inc., Engelwood, NJ, 1981.

Computer Applications Handbook, U.S. Department of Interior, Bureau of Land Management, Handbook No. H-1262-2.

Establishing a Software Engineering Technology, Office of Software Development, General Services Administration, Report No. OSD/FSTC-83/014, June, 1983.

Guidelines for Planning and Implementing a Software Improvement Program (SIP), Office of Software Development, General Services Administration, Report No. OSD/FCSC-83/004, May, 1983.

Life Cycle Management Guidlines, U.S. Department of Interior, Bureau of Land Management, March, 1984.

BLM Library
D-553A, Building 50
Denver Federal Center
P. O. Box 25047
Denver, CO 80225-0047